# Data Structures

Annalise Tarhan

April 13, 2021

## 1 Give an algorithm that returns true if a string of parentheses is properly nested and balanced and false otherwise. Also identify the position of the first offending parenthesis if the string is not properly nested and balanced.

The data structure used to solve this problem is a stack of ints, each representing the position of an open parenthesis. The algorithm is structured as a for loop where $i$ marks the position in the string. If the character at position $i$ is an open parenthesis, $i$ is added to the stack. If it is a close parenthesis, a value is popped from the stack and discarded.

If the stack is empty when a close parenthesis is encountered, the function prints the current position (that of the improper close parenthesis) and returns false. When the end of the string is reached, the stack should be empty. If it isn't, all but the last element is popped and discarded, the last element is printed, recording the position of the first dangling open parenthesis, and the function returns false. If the stack is empty when the end of the input string is reached, the function returns true.

## 2 Give an algorithm that takes a string $S$ consisting of opening and closing parentheses, and finds the length of the longest balanced parentheses in S. The solution is not necessarily a contiguous run of parenthesis from $S$.

This problem only requires two ints, one to track the length of balanced parentheses and one to track the number of currently open parentheses. The algorithm iterates through the input string, incrementing the open_paren counter when it encounters an open parenthesis. When it encounters a close parenthesis,

it checks if open_paren is greater than zero. If it is, open_paren is decremented and the length counter is increased by two. When the end of string marker is reached, it returns the length counter.

## 3 Give an algorithm to reverse the direction of a given singly linked list. The algorithm should take linear time.

The first step is to check if the given list is null or only one element long. If so, simply return the list. Otherwise, use three pointers, prev, cur, and next. Prev initially points to the head of the list, cur to head→next, and next to cur→next, which might be null. (Cur can't be null, since we already checked if the list was only one element long.) First, set prev→next to null, since it will be the last node in the list. Then, enter a while loop, which continues as long as next is not null. In the loop, set cur→next to prev, prev to cur, cur to next, and next to next→next. When the loop is finished, set cur→next to prev and return.

## 4 Design a stack $S$ that supports $S.push(x)$, $S.pop()$, and $S.findmin()$, which returns the minimum element of $S$. All operations should run in constant time.

This implementation uses a linked list of structs, each of which contains an integer item, the minimum value so far in the stack, and a pointer to the next node. For simplicity, stacks are initialized with an initial value.

```
typedef struct stack {
        int item;
        int min;
        struct stack *next;
} stack;
```

```
struct stack *init(int x) {
        stack *S = malloc(sizeof(stack));
        S->item = x;
        S->min = x;
        S->next = NULL;
        return S;
}
```

```
void push(struct stack *S, int x) {
        stack *new_S = malloc(sizeof(stack));
```

```
        new_S->item = x;
        new_S->next = S;
        if (x > S->min) {
                new_S->min = S->min;
        } else {
                new_S->min = x;
        }
        *S = *new_S;
}
```

```
int pop(struct stack *S) {
        int popped = S->item;
        S = S->next;
        return popped;
}
```

```
int findmin(struct stack *S) {
        return S->min;
}
```

# 5 Dynamic Arrays

## 5.1 Consider an underflow strategy that cuts the array size in half whenever the array falls below half full. Give an example sequence of insertions and deletions where this strategy gives a bad amortized cost.

The vulnerability is that the size of the array could hover around a power of two, so many of the insertions cause the array to resize upwards and many of the deletions cause the array to resize downwards. For example, insert four elements into an array. Then, insert another, causing a resize to size eight. Then, remove two elements. The array resizes back to size four. Now insert two more. Every other operation causes a resize and the copying of all elements. Half of the operations are constant time, half are $O(n)$, so the amortized cost is $O(n/2) = O(n)$.

## 5.2 Give a better underflow strategy than that suggested above, one that achieves constant amortized cost per deletion.

A better underflow strategy is to wait longer to resize downwards. For example, if arrays double in size when they are full, they should resize downwards to half size when they are only a quarter full. So, the series of insertions and deletions

in the previous example wouldn't cause a resize downwards because the array would remain at least one quarter full. The number of deletions required to trigger a resize would be proportional to the size of the array, not to a constant, so the amortized cost per deletion would be constant.

## 6 Suppose you seek to maintain the contents of a refrigerator so as to minimize food spoilage. What data structure should you use, and how should you use it?

The food in the fridge should be (mentally) structured as a first-in-first-out queue. When something is inserted into the fridge, it is placed at the end of the queue. When it is time to remove something, the item at the front of the queue, or the oldest item in the fridge, is chosen.

## 7 Work out the details of supporting constant-time deletion from a singly linked list as per the footnote from page 79. Support the other operations as efficiently as possible.

Sentinel, a global variable, marks end of list.

```
typedef struct sing_link_list {
        int item;
        struct sing_link_list *next;
} sll;

struct sing_link_list *Sentinel;
```

A newly initialized list is just a reference to the empty Sentinel node.

```
struct sing_link_list *sll_init() {
        sll *list = Sentinel;
        return list;
}
```

When the last item in a list is deleted, it becomes the Sentinel node, or it becomes the node the Sentinel reference points to. Deleting any other node uses the method outlined in the footnote: the 'deleted' node's item changes to its successor's item and its next node changes to skip the successor node and point to the following node instead.

```
void sll_delete(sll *delete_me) {
        if (delete_me->next == Sentinel) {
                Sentinel = delete_me;
        } else {
                delete_me->item = delete_me->next->item;
                sll *old_next = delete_me->next;
                delete_me->next = delete_me->next->next;
                free(old_next);
        }
}
```

```
struct sing_link_list *sll_search(sll *list, int x) {
        sll *cur = list;
        while (cur != Sentinel && cur->item != x) {
                cur = cur->next;
        }
        if (cur == Sentinel) {
                return NULL;
        } else {
                return cur;
        }
}
```

```
void sll_insert(sll **list, int x) {
        sll *new_list = malloc(sizeof(sll));
        new_list->item = x;
        new_list->next = *list;
        *list = new_list;
}
```

```
struct sing_link_list *sll_succ(sll *node) {
        if (node->next == Sentinel) {
                return NULL;
        } else {
                return node->next;
        }
}
```

```
struct sing_link_list *sll_pred(sll *list, sll *node) {
        if (list == node) return NULL;
        sll *cur = list;
        while (cur->next != node) {
                cur = cur->next;
        }
```

```
        return cur;
}
```

```
int sll_max(sll *list) {
        int max = list->item;
        sll *cur = list;
        while (cur != Sentinel) {
                if (cur->item > max) {
                        max = cur->item;
                }
                cur = cur->next;
        }
        return max;
}
```

```
int sll_min(sll *list) {
        int min = list->item;
        sll *cur = list;
        while (cur != Sentinel) {
                if (cur->item < min) {
                        min = cur->item;
                }
                cur = cur->next;
        }
        return min;
}
```

# 8 Create a data structure with $O(n)$ space that accepts a sequence of Tic-Tac-Toe moves and reports in constant time whether the last move won the game.

This implementation works by keeping two arrays of size n+n+2, one for each player. The first $n$ items represent the number of squares in the $n$th row the player has played, the second $n$ items represent the number of squares in the $n$th column the player has played, and the last two items represent the diagonals. By checking and incrementing that array on each turn, whether or not the move won the game can be determined in constant time.

The board array tracks which squares have been played, preventing illegal moves. Each move returns -1 for an illegal move, 1 for a game-winning move, and 0 otherwise. Turns tracks the number of squares played, so the game ends when the board is full, whether or not someone has won.

```
typedef struct game {
        int n, turns;
        int *x, *o, *board;
} game;
```

```
game *ttt_init(int n) {
        game *g = malloc(sizeof(game));
        g->n = n;
        g->turns = 0;
        g->x = malloc(sizeof(int) * (n+n+2));
        g->o = malloc(sizeof(int) * (n+n+2));
        g->board = malloc(sizeof(int) * (n*n));
        for (int i = 0; i < n+n+2; i++) {
                g->x[i] = 0;
                g->o[i] = 0;
        }
        for (int i = 0; i < n*n; i++) {
                g->board[i] = 1;
        }
        return g;
}
```

```
int x_move(game *g, int row, int col) {
        int n = g->n;

        int board_pos = n*col + row;
        if (!g->board[board_pos]) {
                return -1;
        } else {
                printf("X plays (%i,%i)\n", row, col);
                g->board[board_pos] = 0;
        }

        int row_pos = row;
        int col_pos = col + n;
        int game_over = 0;

        /* Increment row */
        if (g->x[row_pos] == n-1) {
                printf("\nX WINS! Row %i\n\n", row);
                game_over = 1;
        } else {
                g->x[row_pos] = g->x[row_pos] + 1;
        }
```

```
        /* Increment column */
        if (g->x[col_pos] == n-1) {
                printf("\nX WINS! Column %i\n\n", col);
                game_over = 1;
        } else {
                g->x[col_pos] = g->x[col_pos] + 1;
        }

        /* Increment diagonal, if appropriate */
        if (row == col) {
                if (g->x[n+n] == n-1) {
                        printf("\nX WINS!
                                Forward diagonal\n\n");
                        game_over = 1;
                } else {
                        g->x[n+n] = g->x[n+n] + 1;
                }
        } else if (row == n-col-1) {
                if (g->x[n+n+1] == n-1) {
                        printf("\nX WINS!
                                Backward diagonal\n\n");
                        game_over = 1;
                } else {
                        g->x[n+n+1] = g->x[n+n+1] + 1;
                }
        }
        g->turns = g->turns+1;
        if (!game_over && g->turns == n*n) {
                printf("\nNobody wins :(\n\n");
                game_over = 1;
        }
        return game_over;
}
```

```
int o_move(game *g, int row, int col) {
        int n = g->n;

        int board_pos = n*col + row;
        if (!g->board[board_pos]) {
                return -1;
        } else {
                printf("O plays (%i,%i)\n", row, col);
                g->board[board_pos] = 0;
        }
```

```
        int row_pos = row;
        int col_pos = col + n;
        int game_over = 0;

        /* Increment row */
        if (g->o[row_pos] == n-1) {
                printf("\nO WINS! Row %i\n\n", row);
                game_over = 1;
        } else {
                g->o[row_pos] = g->o[row_pos] + 1;
        }

        /* Increment column */
        if (g->o[col_pos] == n-1) {
                printf("\nO WINS! Column %i\n\n", col);
                game_over = 1;
        } else {
                g->o[col_pos] = g->o[col_pos] + 1;
        }

        /* Increment diagonal, if appropriate */
        if (row == col) {
                if (g->o[n+n] == n-1) {
                        printf("\nO WINS!
                                Forward diagonal\n\n");
                        game_over = 1;
                } else {
                        g->o[n+n] = g->o[n+n] + 1;
                }
        } else if (row == n-col-1) {
                if (g->o[n+n+1] == n-1) {
                        printf("\nO WINS!
                                Backward diagonal\n\n");
                        game_over = 1;
                } else {
                        g->o[n+n+1] = g->o[n+n+1] + 1;
                }
        }
        g->turns = g->turns+1;
        if (!game_over && g->turns == n*n) {
                printf("\nNobody wins :(\n\n");
                game_over = 1;
        }
        return game_over;
}
```

```
void play(int n) {
        game *g = ttt_init(n);
        int game_over = 0;

        int rc, row, col;
        time_t t;
        srand((unsigned) time(&t));

        while(!game_over) {
                // X's turn
                row = rand() % n;
                col = rand() % n;
                rc = x_move(g, row, col);
                while (rc < 0) {
                        row = rand() % n;
                        col = rand() % n;
                        rc = x_move(g, row, col);
                }
                if (rc > 0) {
                        game_over=1;
                        break;
                }

                // O's turn
                row = rand() % n;
                col = rand() % n;
                rc = o_move(g, row, col);
                while (rc < 0) {
                        row = rand() % n;
                        col = rand() % n;
                        rc = o_move(g, row, col);
                }
                if (rc > 0) {
                        game_over=1;
                        break;
                }
        }
        free(g);
}
```

## 9 Write a function which, given a sequence of digits 2-9 and a dictionary of $n$ words, reports all words described by this sequence when typed in on a standard telephone keypad.

```
void keypad_words(char *digits, struct dictionary *dict) {
        rec_words("", digits, dict);
}
```

```
void rec_words(
        char *word, char *digits, dictionary *dict) {

        /* Get next digit */
        char digit = digits[0];
        int num = atoi(&digit);
        /* Increment digits to remove first digit */
        digits++;

        /* If digits exhausted, look up word */
        if (num < 2 || num > 9) {
                word++;   // Deals with leading space
                if (search(dict, word) != NULL) {
                        printf("%s\n", word);
                }
                return;
        }

        char buf0[strlen(word) + 2];
        char buf1[strlen(word) + 2];
        char buf2[strlen(word) + 2];
        char buf3[strlen(word) + 2];
        strncpy(buf0, word, strlen(word));
        strncpy(buf1, word, strlen(word));
        strncpy(buf2, word, strlen(word));
        strncpy(buf3, word, strlen(word));

        switch (num) {
                case 2:
                strcat(buf0, "a");
                strcat(buf1, "b");
                strcat(buf2, "c");
                break;
                case 3:
                strcat(buf0, "d");
```

```
                strcat(buf1, "e");
                strcat(buf2, "f");
                break;
                case 4:
                strcat(buf0, "g");
                strcat(buf1, "h");
                strcat(buf2, "i");
                break;
                case 5:
                strcat(buf0, "j");
                strcat(buf1, "k");
                strcat(buf2, "l");
                break;
                case 6:
                strcat(buf0, "m");
                strcat(buf1, "n");
                strcat(buf2, "o");
                break;
                case 7:
                strcat(buf0, "p");
                strcat(buf1, "q");
                strcat(buf2, "r");
                strcat(buf3, "s");
                break;
                case 8:
                strcat(buf0, "t");
                strcat(buf1, "u");
                strcat(buf2, "v");
                break;
                case 9:
                strcat(buf0, "w");
                strcat(buf1, "x");
                strcat(buf2, "y");
                strcat(buf3, "z");
                break;
        }
        rec_words(buf0, digits, dict);
        rec_words(buf1, digits, dict);
        rec_words(buf2, digits, dict);

        if (num == 7 || num == 9) {
                rec_words(buf3, digits, dict);
        }
}
```

## 10 Two strings $X$ and $Y$ are anagrams if the letters of $X$ can be rearranged to form $Y$. Give an efficient algorithm to determine whether strings $X$ and $Y$ are anagrams.

The essence of this question is finding an efficient hash function that gives results unique to each combination of letters. I accomplished this using an array called alpha_primes, which stores 26 unique primes. A more complete implementation would account for capital letters as well.

```
int hash(char *word) {
        int hash = 0;
        for (int i = 0; i < strlen(word); i++) {
                hash += alpha_primes[word[i] - 'a'];
        }
        return hash;
}
```

```
void check_anagrams(char *word1, char *word2) {
        if (strlen(word1) != strlen(word2)) {
                printf("Not anagrams\n");
        }
        int hash1 = hash(word1);
        int hash2 = hash(word2);
        if (hash1 == hash2) {
                printf("Anagrams!\n");
        } else {
                printf("Not anagrams\n");
        }
}
```

## 11 Design a dictionary data structure in which search, insertion, and deletion can all be processed in $O(1)$ time in the worst case. You may assume the set elements are integers drawn from a finite set 1,2,...,$n$ and initialization can take $O(n)$ time.

```
typedef struct array_dict {
        int *dict;
```

```
} array_dict;
```

```
array_dict *array_dict_init(int n) {
        array_dict *dict = malloc(sizeof(array_dict));
        dict->dict = malloc(sizeof(int) * n);
        for (int i = 0; i < n; i++) {
                dict->dict[i] = 0;
        }
        return dict;
}
```

```
void array_dict_insert(array_dict *dict, int x) {
        dict->dict[x] = 1;
}
```

```
int *array_dict_search(array_dict *dict, int x) {
        if (dict->dict[x] == 1) {
                return &(dict->dict[x]);
        } else {
                return NULL;
        }
}
```

```
void array_dict_delete(array_dict *dict, int x) {
        dict->dict[x] = 0;
}
```

## 12 The maximum depth of a binary tree is the number of nodes on the path from the root down to the most distant leaf node. Give an $O(n)$ algorithm to find the maximum depth of a binary tree with $n$ nodes.

This recursive algorithm works by determining the depth of each of its two children, comparing them, and returning the higher value plus one, to account for itself. A leaf node returns one.

```
int find_max_depth(binary_tree *tree) {
        int max = 0;
        if (tree->left_child != NULL) {
                max = find_max_depth(tree->left_child);
        }
```

```
        if (tree->right_child != NULL) {
                int right_depth =
                        find_max_depth(tree->right_child);
                if (right_depth > max) {
                        max = right_depth;
                }
        }
        return max + 1;
}
```

## 13 Two elements of a binary search tree have been swapped by mistake. Give an $O(n)$ algorithm to identify these two elements so they can be swapped back.

The key to identifying an out of place element is tracking the min and max bounds on it. The first element, the root of the tree, is not bounded, but every element to its left must be less than the root and every element to the right must be greater. Each step down the tree tightens these bounds.

To find the two out of place elements in a binary search tree containing integer elements, use the following recursive function. The first binary tree argument initially points to the root of the tree, the next two arguments are initally null, and min and max start at INT_MIN and INT_MAX respectively and are tightened as the tree is traversed. By the time the function returns, first and second will point to the swapped elements.

```
void find_swapped(binary_tree *tree, binary_tree *first,
        binary_tree *second, int min, int max) {

        /* Check if current element is out of place */
        if (tree->item > max || tree->item < min) {
                if (first != NULL) {
                        second = tree;
                        return;
                } else {
                        first = tree;
                }
        }
        /* Inspect left child tree */
        find_swapped(tree->left_child, first, second,
                min, tree->item);
```

```
        /* If necessary, inspect right tree */
        if (second == NULL) {
                find_swapped(tree->right_child, first,
                        second, tree->item, max);
        }
}
```

## 14 Given two binary search trees, merge them into a doubly linked list in sorted order.

This algorithm takes advantage of binary search trees' knack for sorting and begins by (destructively) merging the two trees. Once they are merged, it is simple to traverse the tree from right to left, inserting each item at the beginning of the new list.

```
struct doubly_linked_list *trees_to_list(
        binary_tree *tree1, binary_tree *tree2) {

        merge_trees(tree1, tree2);
        dll *list = dll_init();
        tree_to_list(list, tree1);
        return list;
}
```

```
void merge_trees(binary_tree *tree1, binary_tree *tree2) {
        tree_insert(&tree1, tree2->item);
        if (tree2->left_child != NULL) {
                merge_trees(tree1, tree2->left_child);
        }
        if (tree2->right_child != NULL) {
                merge_trees(tree1, tree2->right_child);
        }
}
```

```
void tree_to_list(dll *list, binary_tree *tree) {
        if (tree->right_child != NULL) {
                tree_to_list(list, tree->right_child);
        }
        insert_at_front(list, tree->item);
        if (tree->left_child != NULL) {
                tree_to_list(list, tree->left_child);
        }
}
```

```
void tree_insert(binary_tree **tree, int item) {
        /* Initialize tree if null */
        if (*tree == NULL) {
                binary_tree *new_tree =
                        malloc(sizeof(binary_tree));
                new_tree->item = item;
                new_tree->left_child = NULL;
                new_tree->right_child = NULL;
                *tree = new_tree;

        /* Not smart enough for duplicates */
        } else if (item == (*tree)->item) {
                printf("Can't insert duplicate items\n");

        /* Make leaf node */
        } else if ((item > (*tree)->item
                && (*tree)->right_child == NULL)
                || (item < (*tree)->item
                && (*tree)->left_child == NULL)) {

                binary_tree *new_node =
                        malloc(sizeof(binary_tree));
                new_node->item = item;
                new_node->right_child = NULL;
                new_node->left_child = NULL;

                /* Insert leaf node to right */
                if (item > (*tree)->item) {
                        (*tree)->right_child = new_node;

                        /* Insert leaf node to left */
                } else {
                        (*tree)->left_child = new_node;
                }

                /* Call function recursively */
        } else if (item > (*tree)->item) {
                tree_insert(&(*tree)->right_child, item);
        } else {
                tree_insert(&(*tree)->left_child, item);
        }
}
```

```
void insert_at_front(dll *list, int item) {
        /* Create new node */
```

```
        dll_node *new_node = malloc(sizeof(dll_node));
        new_node->item = item;
        new_node->prev = NULL;
        new_node->next = list->head;

        /* Insert new node into list */
        if (list->head != NULL) {
                list->head->prev = new_node;
        }
        list->head = new_node;
}
```

# 15 Describe an $O(n)$-time algorithm that takes an $n$-node binary search tree and constructs an equivalent height-balanced binary search tree.

Populate an array of size $n$ by performing an in-order traversal of the tree, which will produce a sorted array. Then create a new binary search tree, recursively inserting the middle element. The recursive function should accept a reference to the sorted array and a pair of indices, initially $(0, n-1)$. It will insert the element at $\lfloor n/2 \rfloor$, then call itself twice, first with $(0, \lfloor n/2 \rfloor - 1)$, then with $(\lfloor n/2 \rfloor + 1, n-1)$. When the function is eventually called with equal indices, the element at that position is inserted and the function returns. This will eventually result in all elements being inserted in the proper order, giving a height-balanced binary search tree in $O(n)$ time.

# 16 Find the storage efficiency ratio for each of the following binary tree implementations on $n$ nodes.

## 16.1 All nodes store data, two child pointers, and a parent pointer. The data field requires 4 bytes and each pointer requires 4 bytes.

Each node uses 4 bytes for data and 12 bytes for pointers. The efficiency ratio is 25%.

### 16.2 Only leaf nodes store data; internal nodes store two child pointers. The data field requires four bytes and each pointer requires two bytes.

In a perfectly balanced tree, where there are exactly $2^n$ leaf nodes for some $n$, there are $2^n - 1$ internal nodes. This gives $4(2^n - 1)$ bytes for pointers and $4(2^n)$ for data, for a ratio of just over 50%. Less perfect trees give similar results.

## 17 Give an $O(n)$ algorithm that determines whether a given $n$-node binary tree is height-balanced.

This algorithm recursively traverses the tree, updating min and max variables when leaf nodes are reached, if appropriate. Min is initially (and arbitrarily) set to 999 and max to 0. A tree with exactly one node will update both. At the end, the min and max values are compared to see if the difference is more than one.

```
void check_depth(binary_tree *tree, int depth,
        int *min, int *max) {

        if (tree->left_child == NULL &&
                tree->right_child == NULL) {

                if (depth < *min) {
                        *min = depth;
                }
                if (depth > *max) {
                        *max = depth;
                }
        }
        if (tree->left_child != NULL) {
                check_depth(tree->left_child, depth+1,
                        min, max);
        }
        if (tree->right_child != NULL) {
                check_depth(tree->right_child, depth+1,
                        min, max);
        }
}
```

```
void is_balanced(binary_tree *tree) {
        if (tree == NULL) {
                printf("Are null trees balanced?\n");
                return;
        }
```

```
        int min = 999;
        int max = 0;
        check_depth(tree, 1, &min, &max);
        if (max - min > 1) {
                printf("Not balanced.");
        } else {
                printf("Balanced!");
        }
        printf(" Max depth is %i, min depth is %i\n",
                max, min);
}
```

## 18 Describe how to modify any balanced tree data structure such that search, insert, delete, minimum, and maximum still take $O(log\ n)$ time each, but successor and predecessor now take $O(1)$ time each. Which operations have to be modified to support this?

Each node will need to record its predecessor and successor in addition to its value, parent, and children. Insertions and deletions will require extra steps. During insertion, the value of the new node should be compared with the predecessor and successor values of the nodes that are encountered. If the new node's value is closer to the old node's value than the recorded predecessor or successor, the new node's place has been found. Each of those nodes will point to the new node instead, and the new node back at them. For example, if 8 is being inserted and encounters a node with value 5 whose previous successor was 9, 8's predecessor will be 5 and its successor will be 9.

Deletions are fairly simple, since the soon to be deleted node will already have references to the nodes that need to be updated. The doomed node's predecessor and successor will need to be modified to point to each other instead.

## 19 Suppose you have access to a balanced dictionary data structure that supports search, insert, delete, minimum, maximum, successor, and predecessor in $O(log\ n)$ time. Explain how to modify the insert and delete operations so they still take $O(log\ n)$ but now minimum and maximum take $O(1)$ time.

It is obvious that the minimum and maximum values in the tree should be recorded somewhere easily accessible and updated if appropriate on each insertion and deletion. The question is where that easily accessible location is. My first thought was the root node, since it is the first node encountered on any search so comparing the new value to the old min and max is straightforward. The problem is that root nodes are subject to deletion like any other node, and because of the recursive nature of tree structures, it isn't known during deletions whether a parent node is actually the root node and converting the new root into a special root node would be tricky.

For this reason, I think it makes more sense to track min and max outside the tree structure itself. I would create a tree struct that contains a pointer to a root node as well as the min and max. Whenever a new node is inserted, min is updated if the new node's value is smaller than the previous min, and similarly if the new node's value is larger than the previous max. When the min or max node is deleted, the tree would need to be traversed to find the new min or max. This wouldn't add much overhead, since deleting the min or max node requires reaching that area anyway and the new min or max would either be a parent or descendant of the previous min or max, and finding it would still be $O(log\ n)$ time. In the case of duplicate elements, the tree struct would also have to track the number of elements with the min and max values, so they wouldn't be updated until all duplicates have been deleted.

## 20 Design a data structure to support the following operations: $insert(x, T)$, $delete(k, T)$, and $member(x, T)$ where $delete(k, T)$ deletes the $k$th smallest element. All operations must take $O(log\ n)$ time on an $n$-element set.

This data structure is a binary search tree where each node tracks how many items there are in its left sub-tree. When an item is inserted, each time it passes to a node's left child the node's num_left is incremented. Deleting an

item traverses the tree, comparing $k$ to num_left. When moving left, the parent node's num_left is updated, but $k$ doesn't change. When moving right, num_left stays the same, but $k$ is decremented by num_left plus the parent node's count. When $k$ equals num_left, the current node needs to be deleted. That is exactly as complicated as it usually is with binary trees.

Note: This implementation isn't perfect. Specifically, it crashes when the root node is deleted. Other bugs are likely. Additionally, it won't meet the time requirement until it is modified to balance itself upon insertion.

```
typedef struct set {
        struct set_node *root;
        int num_items;
} set;
```

```
typedef struct set_node {
        int item, count, num_left;
        struct set_node *left, *right, *parent;
} set_node;
```

```
set *set_init() {
        set *T = malloc(sizeof(set));
        T->root = NULL;
        T->num_items = 0;
        return T;
}
```

```
set_node *node_init(int x, set_node *parent) {
        set_node *node = malloc(sizeof(set_node));
        node->item = x;
        node->count = 1;
        node->num_left = 0;
        node->left = NULL;
        node->right = NULL;
        node->parent = parent;
        return node;
}
```

```
void node_insert(int x, set_node *T) {
        /* For duplicates, increment node's count */
        if (x == T->item) {
                T->count++;

                /* Insert to left */
        } else if (x < T->item) {
```

```
                T->num_left++;
                if (T->left == NULL) {
                        T->left = node_init(x, T);
                } else {
                        node_insert(x, T->left);
                }

                /* Insert to right */
        } else {
                if (T->right == NULL) {
                        T->right = node_init(x, T);
                } else {
                        node_insert(x, T->right);
                }
        }
}
```

```
void set_insert(int x, set *T) {
        if (T->root == NULL) {
                T->root = node_init(x, NULL);
                T->num_items = 1;
        } else {
                T->num_items++;
                node_insert(x, T->root);
        }
}
```

```
int node_member(int x, set_node *T) {
        if (x == T->item) return 1;
        else if (x < T->item) {
                if (T->left == NULL) return 0;
                else return node_member(x, T->left);
        }
        else {
                if (T->right == NULL) return 0;
                else return node_member(x, T->right);
        }
}
```

```
int set_member(int x, set *T) {
        if (T->root == NULL) return 0;
        else return node_member(x, T->root);
}
```

```
set_node *get_next_biggest(set_node *T) {
        set_node *cur = T;
        while (cur->left != NULL) {
                cur = cur->left;
        }
        return cur;
}
```

```
void node_delete(int k, set_node *T) {
        /* Calculate number of items in current node
        plus all nodes to the left */
        int nodes_here_and_left = T->num_left + T->count;

        /* kth node in left subtree */
        if (T->num_left > k) {
                T->num_left--;
                node_delete(k, T->left);

        /* kth node in right subtree */
        } else if (nodes_here_and_left <= k) {
           node_delete(k - nodes_here_and_left, T->right);

        /* kth node is this node */
        } else {
                if (T->count > 1) {
                        T->count--;
                } else {
                /* Need to know if node to be deleted
                is a left or right child */
                        int is_left_child;
                        if (T->parent->left == T) {
                                is_left_child = 1;
                        } else {
                                is_left_child = 0;
                        }

                        /* If node is a leaf,
                        replace it with null */
                        if (T->right == NULL
                                && T->left == NULL) {
                                if (is_left_child) {
                                  T->parent->left = NULL;
                                } else {
                                  T->parent->right = NULL;
                                }
```

```
                free(T);

        /* If node only has one child,
        make parent and child point
        to each other instead */
        } else if (T->right == NULL
        || T->left == NULL) {
                set_node *child;
                if (T->right == NULL) {
                        child = T->left;
                } else {
                        child = T->right;
                }
                if (is_left_child) {
                 T->parent->left = child;
                } else {
                 T->parent->right = child;
                }
                child->parent = T->parent;
                free(T);

        /* If node has two children,
        find next biggest and switch */
        } else {
           set_node *next_biggest
           = get_next_biggest(T->right);

/* When next biggest has a right child */
        if (next_biggest->right != NULL) {
        next_biggest->parent->left
                = next_biggest->right;
        next_biggest->right->parent
                = next_biggest->parent;

 /* When next biggest is a leaf node */
        } else {
                 next_biggest->parent->left
                        = NULL;
        }

/* Make deleted node copy next biggest */
        T->item = next_biggest->item;
        T->count = next_biggest->count;
        free(next_biggest);
        }
}
```

```
        }
}
```

```
void set_delete(int k, set *T) {
        if (k > T->num_items || k < 0) {
                printf("Bad request\n");
        } else if (T->num_items == 1) {
                free(T->root);
                T->num_items = 0;
                T->root = NULL;
        } else {
                node_delete(k, T->root);
        }
}
```

## 21 A concatenate operation takes two sets $S_1$ and $S_2$, where every key in $S_1$ is smaller than any key in $S_2$ and merges them. Give an algorithm to concatenate two binary search trees into one binary search tree. The worst-case running time should be $O(h)$, where $h$ is the maximal height of the two trees.

If we know that all of the elements of one tree are strictly less than all elements of the other tree, it is a simple task to determine which is which, then find the minimum element of the tree with the larger elements. Its left child will be null, since it is the smallest element. Make it's left child pointer point to the root of the other tree instead. No one said it had to be balanced!

Note: This question is marked difficulty 8, which is the highest I've encountered so far. That makes me suspicious of my answer, since it seems so straightforward.

## 22 Design a data structure that supports the following two operations: $insert(x)$ and $median(x)$. All operations must take $O(log\ n)$ on an $n$-element set.

This solution uses a binary search tree that tracks the number of elements and whose nodes track the number of items in their left sub-tree. It supports duplicate elements and works similarly to the tree from two problems ago. To find the median, the tree first calculates which element that will be, rounding up. It then calls the recursive function find_mth_element.

Note: This implementation would only meet the time requirement if it was modified to balance itself on insertion. Upon reflection, once the tree is self-balancing, this question is trivial. The root will always be the median element of a balanced tree.

```
typedef struct median_node {
        int item, count, num_left;
        struct median_node *left, *right;
} median_node;
```

```
typedef struct median_tree {
        int num_items;
        median_node *root;
} median_tree;
```

```
median_tree *median_init() {
        median_tree *tree = malloc(sizeof(median_tree));
        tree->num_items = 0;
        tree->root = NULL;
        return tree;
}
```

```
median_node *median_node_init(int x) {
        median_node *node = malloc(sizeof(median_node));
        node->item = x;
        node->left = NULL;
        node->right = NULL;
        node->count = 1;
        node->num_left = 0;
        return node;
}
```

```
void median_node_insert(int x, median_node *node) {
        if (node->item == x) {
                node->count++;
        } else if (node->item > x) {
                node->num_left++;
                if (node->left == NULL) {
                        node->left = median_node_init(x);
                } else {
                        median_node_insert(x, node->left);
                }
        } else {
                if (node->right == NULL) {
                        node->right = median_node_init(x);
                } else {
                        median_node_insert(x,node->right);
                }
        }
}
```

```
void median_insert(int x, median_tree *tree) {
        tree->num_items++;
        if (tree->root == NULL) {
                tree->root = median_node_init(x);
        } else {
                median_node_insert(x, tree->root);
        }
}
```

```
int find_mth_node(median_node *n, int m) {
        /* Median in left subtree */
        if (m <= n->num_left) {
                return find_mth_node(n->left, m);

                /* Median at current root */
        } else if (m <= n->num_left+n->count) {
                return n->item;

                /* Median in right subtree */
        } else {
                int num_passed = n->num_left + n->count;
                return find_mth_node(
                        n->right, m-num_passed);
        }
}
```

```
int median(median_tree *tree) {
        if (tree->num_items == 0) {
                printf("Bad request, tree is empty!\n");
                exit(1);
        } else {
                int median = (tree->num_items / 2) + 1;
                return find_mth_node(tree->root, median);
        }
}
```

## 23 Assume we are given a standard dictionary, a balanced binary search tree, defined on a set of $n$ strings, each of length at most $l$. We seek to print out all strings beginning with a particular prefix $p$. Show how to do this in $O(ml\ log\ n)$ time, where $m$ is the number of strings.

Any prefix corresponds to a range of strings beginning with the prefix itself and ending (non-inclusively) with the last letter of the prefix switched to the next letter in the alphabet. For example, the prefix 'pre' corresponds to all strings between 'pre' and 'prf.' The algorithm to find all such strings begins at the root and determining if the string is in the range or not. If it isn't, the traversal continues to the appropriate sub-tree. If the node's string is within the range, its string is printed and both of its sub-trees are explored.

The time complexity of comparing two strings is bounded by $l$, the maximum length of the strings, so finding the part of the tree that contains strings in the relevant range is bounded by $l * log(n)$. Once that area is discovered, the time complexity of discovering all the matching strings is on the order of $lm$. It seems to me that the total time complexity is more like $O(lm + l\ log\ n)$ than $O(ml\ log\ n)$, but of course anything that has $lm$ or $l * log(n)$ as an upper bound also has $lm\ log\ n$ as an upper bound.

## 24 An array $A$ is called $k$-unique if it does not contain a pair of duplicate elements within $k$ positions of each other. Design a worst-case $O(n \log k)$ algorithm to test if $A$ is $k$-unique.

Use a balanced binary search tree whose size never exceeds $k$ items and two index values, $i$ and $j$ to traverse $A$. $i$ begins immediately, and the values it encounters are inserted into the tree. If it encounters the value it is trying to insert, the algorithm immediately returns false. After $i$ reaches the $k$th item of $A$, $j$ begins, starting at $A[0]$. That value is then removed from the tree. This guarantees that only the most recent $k$ items are present in the tree. If $i$ reaches the end of $A$ without encountering a duplicate, $A$ is $k$-unique. Each of the $n$ elements of $A$ corresponds to an insertion and a deletion in the tree, each of which costs at most $\log k$, for a total time complexity on the order of $O(n \log k)$.

## 25 In the bin-packing problem, we are given $n$ objects, each weighing at most one kilogram. Our goal is to find the smallest number of bins that will hold the $n$ objects, with each bin holding at most one kilogram.

### 25.1 Design an algorithm that implements the best-fit heuristic in $O(n \log n)$ time.

Use a balanced binary search tree with the amount of empty space remaining as the key. To place an object, attempt to locate a bin with exactly the object's weight free. The route to the best-fit bin may involve traversing nodes that don't have enough space. In case that path doesn't lead to a bin with adequate space, keep a reference to the most recent 'safe' bin, and if the risky path fails, choose the safe bin. If a safe bin is never found, start a new bin and insert it into the tree. If a safe bin is found, remove it from the tree, add the object's weight, and re-insert it. Each of the $n$ objects requires at most a search, a deletion, and another insertion, each taking $\log n$ time, for a total time equal to $O(n \log n)$.

### 25.2 Repeat the above using the worst-fit heuristic.

This time, simply locate the bin with the most space. If the item fits, delete, update, and reinsert the bin as before. As an optimization, keep a pointer to the bin with the most space. It will need to be updated every time an item fits and changes the bin's value, but it saves a traversal each time the item doesn't fit

and needs to start a new bin. This effectively turns the structure into a priority queue.

## 26 Suppose that we are given a sequence of $n$ values $x_1, x_2, ..., x_n$ and seek to quickly answer repeated queries of the form: given $i$ and $j$, find the smallest value in $x_i$, $x_j$.

### 26.1 Design a data structure that uses $O(n^2)$ space and answers queries in $O(1)$ time.

This data structure takes the form of a two dimensional $n$x$n$ array, $A$, although only just over half of the space is actually used. It enforces the invariant that $i \leq j$. If queries where $j$ is larger than $i$ are possible, switch the values before querying. The first array, $A[0]$, corresponding to $j_1$, consists of $[x_1, min\{x_1, x_2\}, min\{x_1, x_2, x_3\}, ...,$
$min\{x_1, x_2, ..., x_n\}]$. The second array has an invalid first element followed by $[x_2, min\{x_2, x_3\}, ..., min\{x_2, x_3, ..., x_n\}]$. The final array, $A[n-1]$, has only one valid element, $x_n$, in the final position. To answer a query for $i, j$, return $A[i-1, j-1]$. (Or, if $j > i$, $A[j-1, i-1]$.) The time to set up the data structure is $O(n^2)$, but queries are answered in constant time.

### 26.2 Design a data structure that uses $O(n)$ space and answers queries in $O(log\ n)$ time.

Use a balanced binary search tree to store the minimum over a range. The root of the tree contains the global minimum, as well as the index where that minimum is found. It also contains pointers to each of its children. Its left child contains the minimum value in the range $(1, \lfloor n/2 \rfloor)$ and that minimum value's location. The root's right child contains the minimum over $(\lfloor n/2 \rfloor + 1, n)$. (When $n/2$ is a whole number, the right child should start with $n/2 + 1$. An even better solution would use round-to-even or something similar to avoid unbalancing the tree.) The pattern continues, which each of the four nodes on the next level containing the minimum over a quarter of the total range. The leaves contain individual values. Ranges are implicit, based on $n$ being known in advance. A query returns when it reaches a node where the minimum value's index is within query parameters. A traversal might fork when the given range is split between two or more nodes. In that case, the query would return the minimum of the two recursive queries.

This data structure uses $2n - 1$ nodes, each with two data fields and at most two pointers, for a space complexity of $O(n)$. Even the worst-case query, $i = j$ where $x_i$ is the global maximum, only takes $log(n)$ time, since the tree's depth is $log(n)$.

**27 Suppose you are given an input set $S$ of $n$ integers, and a black box that if given any sequence of integers and an integer $k$ instantly and correctly answers whether there is a subset of the input sequence whose sum is exactly $k$. Show how to use the black box $O(n)$ times to find a subset of $S$ that adds up to $k$.**

First, enter the entire set, along with $k$, to confirm that such a subset exists. Then, eliminate one element at a time. If the black box returns that there is a subset adding up to $k$ still exists, eliminate that element from consideration. After $n + 1$ rounds, the elements remaining will be a subset that adds up to $k$. If there is more than one possible subset, the result will depend on the order in which the elements are eliminated.

**28 Let $A[1..n]$ be an array of real numbers. Design an algorithm to perform any sequence of the following operations: $Add(i, y)$, which adds $y$ to the $i$th number, and $Partial-sum(i)$, which returns the sum of the first $i$ numbers. There are no insertions or deletions; the only change is to the values of the numbers. Each operation should take $O(log\ n)$ steps. You may use one additional array of size $n$ as a work space.**

Use an (array-based) balanced binary search tree with $n$ nodes. The key of each node, which is used to order and balance the tree, is the array index. The value of each node is the partial sum of the array, up to and including the number at the node's key's index in the array. For example, the root contains the partial sum of the first $n/2$ numbers. Every other node contains the sum of the numbers represented by the nodes in its left sub-tree. Each node should also include a reference to its counterpart in the original array.

To find the partial sum for any index $i$, traverse the tree starting at the root and maintain an accumulator variable initialized to zero. When the current node's key is greater than $i$, move to the left child, leaving the accumulator unchanged. When $i$ is greater, add the current node's value to the accumulator variable and

move to the right child. When a node is found whose key equals $i$, add its value to the accumulator and return the accumulated value.

To add $y$ to the $i$th number, start at the root. If the root's key is greater than $i$, add $y$ to its value and move to the left sub-tree. If the root's key is smaller, leave its value unchanged and move to the right sub-tree. If the key equals $i$, add $y$ to it, update its corresponding array element, and return.

**29 Extend the data structure of the previous problem to support insertions and deletions. Each element now has both a key and a value. An element is accessed by its key, but the addition operation is applied to the values. The worst-case running time should still be $O(n\ log\ n)$ for any sequence of $O(n)$ operations. The data structure should support the following operations:**
**$Add(k, y)$ - adds $y$ to the item with key $k$,**
**$Insert(k, y)$ - inserts a new item $(k, y)$**
**$Delete(k)$ - deletes the item with key $k$**
**$Partial - sum(k)$ - returns the sum of all elements whose key is less than $k$.**

Although this extension adds quite a bit of functionality, it changes very little about the underlying data structure. As long as the keys are unique and have a natural ordering, it makes no difference whether they are contiguous non-negative integers or arbitrary values.

Add works the same way as before: start at the root and compare the root's key with the $k$ parameter. If it is equal, add $y$ to the previous value. If it is smaller, add $y$ to the root's partial-sum value, move to the left child, and repeat. If it is larger, leave the root's partial-sum value unchanged and move to the right. An important difference is that when the node with key $k$ is found, its value is changed but not it's partial-sum, since in this version, an element's partial-sum value does not include its own value.

Insert works much the same way, traversing the tree from the root and adjusting the partial values in nodes it encounters on its way to their left children. Instead of locating a node with the same $k$ value, it inserts a new leaf node at the appropriate location.

Delete requires two traversals: the first to determine the $y$ value of the node and the second to subtract that value from the appropriate partial sums on the way down. After the second traversal, the node is deleted and the tree re-balanced.

As before, $Partial - sum$ is simple, since all the other operations do the real work. Simply locate the node with key $k$ and return its partial-sum value.

## 30 You are consulting for a hotel that has $n$ one-bed rooms. When a guest checks in, they ask for a room whose number is in range $[l, h]$. Propose a data structure that supports the following data operations in the allotted time:

$Initialize(n)$ **- initialize data structure in polynomial time**

$Count(l, h)$ **- return the number of available rooms in** $[l, h]$ **in** $O(log\ n)$ **time**

$Checkin(l, h)$ **- in** $O(log\ n)$ **time, return the first empty room in** $[l, h]$**, mark it occupied, or return NIL if all rooms are occupied**

$Checkout(x)$ **- mark room** $x$ **as unoccupied, in** $O(log\ n)$ **time**

This solution uses the same idea as problem 26.2, which used a balanced binary tree to return the smallest value in any given range. It accomplishes this by having each node represent a range instead of a specific element. The leaf nodes represent a range of one, or the individual rooms. Each node contains a reference to the first open room in its range as well as the total number of open rooms in its range.

Initialization creates a balanced binary tree of depth $\lceil log_2(n) \rceil + 1$. Nodes do not have keys in the normal sense. Instead, they have two 'keys' that work together: the first and last rooms in their range. The root node represents

the entire range $(1, n)$, containing a reference to the first unoccupied room and recording the total number of unoccupied rooms at the hotel. Determining the minimum over each range is simple for initialization. All rooms are unoccupied, so the minimum is just the first room in a node's range.

*Count* will usually traverse multiple paths down the tree, since it needs to find all unoccupied rooms in the range. For example, if there is a hotel with eight rooms (1,8) and the requested range is (3,6), both of the root's children will be called. The left child, which represents (1,4) will be called with $Count(3, 4)$ and the right child will be called with $Count(5, 6)$. Because neither set of arguments exactly represents the child nodes' range, there is one more set of calls to be made. The $(1, 4)$ node's right child will be called with $Count(3, 4)$, which can return immediately since that child's range is exactly $(3, 4)$. Similarly at the root's right child, whose range is $(5, 8)$, the call will be made to its left child with $Count(5, 6)$. Again, the traversal ends there because the arguments exactly match the range. The original function will return the sum of the calls to its children.

The time complexity for *count* is more complicated than usual for a binary tree, but I believe it is still $O(log\ n)$, since the input range must be continuous. The effect of that continuity is that there are only two traversals: one to the minimum element in the range and one to the maximum. For any given request, each node encountered will either be an inner node, where its entire range is a subset of the requested range, or an outer node, where only part of its range is a subset of the requested range. Inner nodes can return immediately with the total number of open rooms in their range. Only outer nodes need to call their children, since their open room count may include rooms outside the requested range. Of those children, there can be at most one outer node, for the same reason as before. (Unless this node's range contains the entire requested range, so this node functions as the root node and the traversal splits here.) So, even if each of the traversal paths calls two child nodes at each level (for a total of four at each level), at least two of them will be inner nodes and dead end. The more relevant fact is that at most two of the nodes can be expensive outer nodes. This gives a time complexity bounded by $4log\ n$, which is still $O(log\ n)$.

*Checkin*'s traversal pattern is similar to *count*'s, but a node will only call *checkin* on its right child if its left child returns NIL. As always, traversal begins at the root. If the root's minimum free room is within the requested range, that is the room that will be returned. (But not yet!) Otherwise, if the requested range includes any rooms in the first half of the root node's range, it calls *checkin* on its left child. If that returns anything except NIL, it decrements its open room count and returns the chosen room. If the left child returns NIL, it calls the right child, assuming the right child also includes part of the requested range. If it doesn't, or if calling it returns NIL, the parent node also returns NIL with no further changes.

Once the first unoccupied room in the requested range is located, a different function is called to traverse the rest of the path to the room's leaf node. The room is identified as soon as it appears as a node's minimum, and once it is occupied, that node, and all the others between it and the leaf node, will need a new minimum. This function, call it *checkin*2, doesn't return a reference to the chosen room, but to the next open room in the range. On the way down, all the function does is identify the appropriate child node and call *checkin*2 on it. When the leaf node is reached, it is marked occupied and *checkin*2 returns NIL to the parent node. On the way back up, *checkin*2 decrements the open room count and updates the node's minimum room reference. If the *checkin*2 call to the node below it returned a room number, that room number will be the new minimum. If it returned NIL, the new minimum will be the other child's minimum, which might also be NIL. Once the last *checkin*2 call returns, the *checkin* function calls continue the traversal back up to the root node, which returns the room number, or NIL if one wasn't found.

Happily, checkout is simple. Beginning at the root, the open room count is incremented and if the newly available room's number is smaller than the previous minimum, its number replaces the old one. The traversal continues with exactly the same pattern until the room's leaf node is reached, where the room is marked available. These calls don't return anything, so once the leaf node is reached, all function calls return immediately.

## 31 Design a data structure that allows one to search, insert, and delete an integer $X$ in $O(1)$ time. Assume that $1 \leq X \leq n$ and that there are $m + n$ units of space available, where $m$ is the maximum number of integers that can be in the table at any one time. Use two arrays, $A[1..n]$ and $B[1..m]$. The arrays cannot be initialized, so they are full of garbage.

I was convinced that the question as asked is impossible, and a few minutes of googling hasn't dissuaded me. But, the difference between 'impossible' and 'solved decades ago' seems to be one integer's worth of space and a single initialization operation. All credit to https://research.swtch.com/sparse, which explains how to use a counter, initialized to zero, to indicate how many valid entries there are in $B$ and guarantee accuracy. The problem doesn't say whether duplicates should be allowed, but I've chosen to allow exactly one copy to exist within the data structure.

To search for $X$, determine if $A[X]$ points to a valid entry in $B$ and if that entry of $B$ points back to $A[X]$. First, make sure that $A[X] < count$, since only the first *count* entries of $B$ are valid. Then, if $B[A[X]] == X$, $X$ is present.

To insert $X$, first check if it is already present using *search*. If it isn't, set $A[X] = count$, $B[count] = X$, and increment *count*.

Deleting an element is trickier, since $B$ can't have any holes in its valid section. Plug the gap by switching in the last valid element of $B$ and updating the corresponding element of $A$. Updating $A[X]$ isn't necessary, since its data is assumed to be garbage. So, find the hole in $B$ and fill it by setting $B[A[X]] = B[count-1]$. Then, update $A$ to keep the swapped in element's entries pointing at each other: $A[B[count - 1]] = A[X]$. Finally, decrement *count*.

## 32 Implement versions of several different dictionary data structures, such as linked lists, binary trees, balanced binary search trees, and hash tables. Conduct experiments to assess the relative performance of these data structures in a simple application that reads a large text file and reports exactly one instance of each word that appears within it. Write a brief report with your conclusions.

I measured the time to create and populate the data structures separately from the time it took to traverse the structure and print the contents. Unsurprisingly, the linked list took by far the longest to create. I did find it surprising that reporting its contents was also the slowest, although only twice as slow as the next slowest.

Another surprise was that the balanced binary tree took much longer to create than the unbalanced one. It did report the results slightly faster, but it seems that the cost of the rebalancings outweighed the performance gain from a more balanced tree.

The hash table's performance, on the other hand, was completely predictable. Once a reasonable table size and hash function were chosen, its performance was significantly faster than the trees' for construction and even a bit faster for reporting.

| | Linked List | Binary Tree | Balanced Tree | Hash Table |
|---|---|---|---|---|
| Creation | 76,935 | 6,425 | 10,190 | 4,973 |
| Reporting | 11,760 | 4,244 | 4,043 | 3,849 |

## 33 A Caesar shift is a very simple class of ciphers for secret messages. Unfortunately, they can be broken using statistical properties of English. Develop a program capable of decrypting Caesar shifts of sufficiently long texts.

For simplicity, this program only works for lowercase input. The algorithm is simple; it just looks for the letter that appears the most frequently and assumes it represents 'e.' The relevant data structure is an array, indexed from 'a.'

```
void decrypt(char *file_name) {
        /* Letters array used to track how many times
                each letter appears */
        int letters[26];
        for (int i = 0; i < 26; i++) {
                letters[i] = 0;
        }

        /* Read file, count letters */
        FILE *fp = fopen(file_name, "r");
        char *line = NULL;
        size_t len = 0;
        ssize_t nread;
        while ((nread = getline(&line, &len, fp)) != -1) {
                for (int i = 0; i < nread; i++) {
                        char cur = line[i];
                        if (cur >= 'a' && cur <= 'z') {
                                letters[cur - 'a']++;
                        }
                }
        }

        /* Find most frequent letter */
        int most_frequent = 0;
        for (int i = 1; i < 26; i++) {
                if (letters[i] > letters[most_frequent]) {
                        most_frequent = i;
```

```
                }
        }

        /* Most frequent letter probably represents 'e' */
        int shift = most_frequent + 'a' - 'e';
        printf("Most frequent letter is %c,
        so the shift is probably %i\n\n",
        most_frequent + 'a', shift);

        /* Re-read file and print out decrypted version */
        fseek(fp, 0, SEEK_SET);
        while ((nread = getline(&line, &len, fp)) != -1) {
                for (int i = 0; i < nread; i++) {
                        char cur = line[i];

                /* Find, shift, and print letters */
                if (cur >= 'a' && cur <= 'z') {
                        char shifted = cur - shift;
                        if (shifted < 'a') {
                                printf("%c", shifted+26);
                        } else if (shifted > 'z') {
                                printf("%c", shifted-26);
                        } else {
                                printf("%c", shifted);
                        }

                /* Just print non-letters */
                } else {
                        printf("%c", cur);
                }
                }
        }
        fclose(fp);
}
```

# 34 What method would you use to look up a word in a dictionary?

First, make a rough guess at the word's location in the dictionary based on its first letter. On that page, use whichever entry you see first to decide on the next step. Notice whether the entry precedes or follows the word you're trying to look up, as well as how far off it is. Make another guess based on that information and repeat. When you're close, use both the first and last words on a page to determine if the word is present on that page and adjust by a page

or two if necessary. When you've found the right page, repeat the process in miniature: make a guess based on whether the word is closer to the first word on the page or the last, then narrow it down either by choosing a word in the middle of the range or closer to one side or the other if you can quickly guess which it would be closer to.

At every step, the goal is to move quickly, not perfectly. Taking the time to do conscious calculations is inefficient, since additional steps are cheap and any reasonably fast calculation won't be very precise anyway. As opposed to a computer, humans tend to be good at imprecise estimates, so don't use the computer's strategy of splitting the difference, use your intuition to adjust guesses based on your sense of distance between words and how populated different letters' sections are.

## 35 Imagine you have a closet full of shirts. What can you do to organize your shirts for easy retrieval?

Getting dressed should be a creative process, supported by the fact that I seem to have an entire closet just for shirts. Clearly, I care about fashion. The goal then is not to find a specific shirt, but to make it easy to consider my options and choose the one that appeals most in the moment, possibly based on other pieces of my outfit that I've already chosen. Ahead of time, I will be aware of my plans for the day and have a general idea of the weather, so I will only be considering shirts suited to the level of formality and the location of the day's events. The level of formality is more important than warmth for shirts, since I am a fashion conscious person with an appropriate collection of outerwear, presumably in a separate closet. So, I would organize my shirts by levels of formality first, coverage second, then color third, because an aesthetically pleasing closet is conducive to creativity.

## 36 Write a function to find the middle node of a singly linked list.

In a list with an even number of nodes, there will be two middle nodes. This function returns the second one. The strategy is to use two pointers, one moving two nodes at a time and the other moving one at a time. When the first pointer reaches the end, the second will be at the middle.

```
linked_list_node *find_middle(linked_list *list) {
        linked_list_node *fast = list->head;
        linked_list_node *slow = list->head;
        while (fast->next != NULL) {
```

```
                fast = fast->next;
                if (fast->next != NULL) fast = fast->next;
                slow = slow->next;
        }
        return slow;
}
```

## 37 Write a function to determine whether two binary trees are identical.

```
int check_identical(binary_tree *t1, binary_tree *t2) {
        if (t1 == NULL && t2 == NULL) return 1;
        if (t1 == NULL || t2 == NULL) return 0;
        if (t1->item != t2->item) return 0;
        return (check_identical(t1->left, t2->left)
        && check_identical(t1->right, t2->right));
}
```

## 38 Write a program to convert a binary search tree into a linked list.

```
llist *give_tree_get_list(binary_tree *tree) {
        llist *list = malloc(sizeof(llist));
        list->item = tree->item;
        list->next = NULL;
        if (tree->left != NULL) {
                list->next=give_tree_get_list(tree->left);
        }
        if (tree->right != NULL) {
                llist *cur = list;
                while (cur->next != NULL) {
                        cur = cur->next;
                }
                cur->next=give_tree_get_list(tree->right);
        }
        return list;
}
```

## 39 Implement an algorithm to reverse a linked list. Then do it without recursion.

```
llist *reverse_helper(llist *reversed, llist *rest) {
        if (rest == NULL) return reversed;
        llist *new_rest = rest->next;
        rest->next = reversed;
        return reverse_helper(rest, new_rest);
}
```

```
llist *reverse_rec(llist *list) {
        return reverse_helper(NULL, list);
}
```

```
llist *reverse_no_rec(llist *list) {
        llist *prev = NULL;
        llist *cur = list;
        llist *next = list->next;
        while (next != NULL) {
                cur->next = prev;
                prev = cur;
                cur = next;
                next = next->next;
        }
        cur->next = prev;
        return cur;
}
```

## 40 What is the best data structure for maintaining URLs that have been visited by a web crawler? Give an algorithm to test whether a given URL has already been visited, optimizing both space and time.

The two contenders are a hash table and a binary tree. Binary trees are simple in that the size of the data set does not need to be known or estimated ahead of time, no hash function needs to be chosen, and there is no decision to be made in regard to collisions. They are also space efficient. The only drawback is that insertions and searches usually take $O(log\ n)$ time, while hash tables can sometimes achieve $O(1)$ insertions and searches. Even with all of the drawbacks, I conclude that a hash table is the better option. Care would need to be taken in

choosing the size of the table and hash function, and as the data set grew the table would need to be resized. The cost of resizing would be high, but not so bad after amortization, assuming the resizings were timed appropriately. Since chaining uses extra memory for pointers, I would opt to use open addressing instead.

To test whether a given URL has been visited, calculate the hash of the URL and search in the corresponding bucket. If the bucket is empty, the URL hasn't been visited. If it contains a different URL, look in the next highest bucket and repeat until the either the given URL or an empty bucket is found.

## 41 You are given a search string and a magazine. You seek to generate all the characters in the search string by cutting them out from the magazine. Give an algorithm to efficiently determine whether the magazine contains all the letters in the search string.

Because this problem appears to be about a human perusing a physical magazine, the bottleneck is the time it takes the person to process the words he or she is reading and determine if it contains any of the characters in the search string. To that end, I would start with the cover of the magazine, the titles of the articles, and any other text that is larger and clearer than the main text. This human is also probably pretty good at identifying the less frequent letters, so I would have them consciously look for the Q or the J, but also have the rest of the search string in mind and grab them as they see them. As they find the infrequent letters, they should start looking for the next least frequent. Hopefully, this informal search will have been quick and effective; only a few letters should remain. From there, the search would become more systematic: only look for the first character in the string and search all the available text until it is found. Repeat for the next character. If an entire pass has been made of the magazine without finding one of the letters, we can conclude that the magazine doesn't contain the entire string.

For a more purely algorithmic approach, I would disregard text size and search systematically from the beginning. But first, I would split the search string into two piles: frequent letters and infrequent letters. To make the distinction, any letter worth more than five scrabble points is a good candidate for 'infrequent.' A linked list would work well for each of the piles. As the text is searched, I would compare each letter to the first letter in the frequent pile and every letter in the (hopefully small) infrequent pile. If there is a match, cut the letter out of the magazine and remove it from the pile. Continue until a letter has been

at the head of a list for an entire loop through the magazine, or until the whole string has been found.

## 42 Reverse the words in a sentence. Optimize for time and space.

Read the sentence, turning each word into a linked list node and inserting it at the head of a previously empty singly linked list. When the end of the sentence is reached, return the head of the list or print each node in order, starting with the head.

## 43 Determine whether a linked list contains a loop as quickly as possible without using any extra storage. Also, identify the location of the loop.

Assuming this can be done destructively, change the item of each encountered node to something outside the normal range. If an item that already has that value is encountered, it is the location of a loop. If NULL is reached before a node with that value, there is no loop.

## 44 You have an unordered array $X$ of $n$ integers. Find the array $M$ containing $n$ elements where $M_i$ is the product of all integers in $X$ except for $X_i$. You may not use division. You can use extra memory.

This solution requires an additional $n$ space and produces a solution after two passes through the array. On the first pass, the new array $P$ is populated by a forward product, the product of each element encountered so far including the current one. $P_0$ contains $X_0$, $P_1$ contains $X_0 * X_1$, and so on. When the end of the array is reached, $M_{n-1}$, the last element of $M$, takes the value at $P_{n-2}$, the product of every element of $X$ except the last one. Then the arrays are traversed in the other direction and the values of $P$ are overwritten by a reverse product. $P_{n-1}$ takes the value of $X_{n-1}$, $P_{n-2}$ takes $X_{n-1} * X_{n-2}$, and so on. At each step, $M_i$ is calculated as the product of $P_{i+1} * P_{i-1}$, where $P_{i+1}$ is the reverse product just calculated and $P_{i-1}$ is the forward product about to be overwritten. When $M_0$ is reached, it takes the value of $P_1$. The two tables below show the state after the forward pass and after the reverse pass.

| X | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| M | | | | 6 |
| P | 1 | 2 | 6 | |

| X | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| M | 24 | 12 | 8 | 6 |
| P | 1 | 24 | 12 | 4 |

## 45 Give an algorithm for finding an ordered word pair occurring with the greatest frequency in a given webpage. Which data structures would you use? Optimize both time and space.

I would use a composite data structure: a hash table where each entry points to a binary search tree. The first word in each word pair would be the key for a hash table entry and the second word would be the key for a node in the tree the hash table entry points to. Each node in the tree would also contain a counter recording how many times that phrase has occurred so far. Since there will be no deletions, the hash table should use open addressing. Additionally, there would be a global variable recording the most frequent phrase so far and another indicating how many times that phrase has appeared.

For the example 'New York,' first locate the entry for 'new' in the hash table, then insert 'york' into the tree it references. If this is the first instance of 'New York,' initialize a node in the 'new' tree with 'york' as the key and one as the count. If 'york' already exists in the tree, locate its node, increment the count, and compare the new count with the global max count. If the new count is greater, update the global variable to 'New York' and the new count. At the end of the webpage, simply return whatever phrase is stored in that variable.

The reason for using a hash table is that insertion is fast and it isn't excessively memory-intensive if the size and hash function are well chosen. So why not have each hash table entry point to another hash table for the second word? Because most words won't appear frequently and almost all of the space allocated for those hash tables would be wasted. A binary tree, on the other hand, only uses as much space as needed, and insertion is still reasonably fast, especially in small trees.