CS1 Review

Alexander Paolini

April 24, 2025

CS1 is the introductory Data Structures and Algorithms Course at UCF. It uses the C programming language and covers basic content.

This overview assumes you, the reader, already know basic C.

At the end of some sections contains LeetCode Questions that can be done for practice.

- Alexander Paolini

Contents

Memory Management in C

Memory Management in C is a review from `COP3223: Intro to C`.

Memory management in C is very simple with regards to this class.

1. `malloc(size_t n)` - Allocates n bytes of memory
2. `free(void *ptr)` - Frees the memory at the pointer specified

Example 1:

```
int main()
{
    // Allocate memory
    int *x = malloc(sizeof(int));

    *x = 5;
    printf("x = %d\n", *x);

    // Free the allocated memory
    free(x);

    return 0;
}
```

`Note`: You would never malloc a primative data type like this.

Example 2: Allocating a student struct and then doing something with said struct.

```
// Basic student struct
typedef struct student_s
{
    char *name;
    double gpa;
} student_t;

void print_student(student_t *s);

int main()
{
    // Allocate memory for student
    student_t *student = (student_t *)malloc(sizeof(student_t));

    // Allocate memory for name; assign name
    student->name = (char *)malloc(sizeof(char) * 20);
    student->name = "Alex";

    print_student(student);

    // Memory must be freed inside-out
    free(student->name);
    free(student);

    return 0;
}
```

Order Analysis

Order Analysis is the way in which we categorize the efficienty of algorithms in terms of its input.

We generally use Big-Oh notation, where the runtime of a piece of code is $O(x)$, which is said as "Big-Oh of x", or "Order x".

Common runtimes include:

1. $O(1)$ - Constant runtime
2. $O(\log(n))$ - Logarithmic runtime
3. $O(n)$ - Linear runtime
4. $O(n\log(n))$ - Linearithmic runtime
5. $O(n^2)$ - Quadratic runtime
6. $O(n^c)$ - Polynomial time complexity
7. $O(2^n)$ - Exponential time complexity
8. $O(n!)$ - Factorial

In Big-Oh notation, we focus only on the general shape of the growth curve, ignoring constants and lower-order terms. For instance, if a runtime is expressed as $T_1n^3 + T_2n^2 + T_1$, we simplify it to $O(n^3)$. This is because the highest-degree term dominates the growth, and Big-Oh captures only the rough upper-bound behavior, not the exact details.

Example 1: Find the Big-Oh notation of a function with the runtime of $T(n) = 5n^2 + 3n + 10$

We ignore the coefficients and lower-order terms, focusing only on the dominant term. The highest-order term is n^2, so the Big-Oh notation is: $O(n^2)$

Example 2: Find the Big-Oh notation of a function with the runtime of $T(n) = 4n^3 + 100nlogn + 600$

Again, the n^3 term dominates the growth as n gets large. So the Big-Oh notation is: $O(n^3)$

Summations

Summations, with respect to this class, are a way to construct runtime. They are defined with the sigma character $\sum$.

Example: The sum of all integers from 0 to 10 – $\sum_{i=0}^{10} i$

Summations are very similar to for loops in programming:

```
int sum = 0;
for (int i = 0; i <= 10; i++)
    sum += i;
```

Example 1: Use summations to find out how many times this print statement executes

```
for (int i = 0; i < 5; i++)
    printf("i = %d\n", i);
```

This starts at 0; Goes to 4; and adds 1 each iteration. So we can say it runs $\sum_{i=0}^{4} 1 = 4$ times.

Closed Form Solutions of Summations

Summations can be solved to create a closed form. This is how we can translate from our for loops to Big-Oh notation. Some basic solutions include:

1. `Arithmetic Series` - $S_n = \frac{n}{2}(2a + (n-1)d)$.
2. `Sum of Squares` - $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.
3. `Sum of Cubes` - $\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2$.
4. `Geometric Series` - $S_n = \frac{a(1-r^n)}{1-r}$.

Summation Manipulations

Oftentimes we will need to manipulate our summations to create closed form solutions that we already know.

1. $\sum ca_k = c \sum a_k$
2. $\sum (a_k + b_k) = \sum a_k + \sum b_k$
3. $\sum a_k x^{i+k} = x^i \sum a_k x^k$
4. $\sum_m^n a_{k+i} = \sum_{m+i}^{n+i} a_k$
5. $\sum_1^n (a_k - a_{k-1}) = a_n - a_0$ or $\sum_1^n (a_{k-1} - a_k) = a_0 - a_n$

Algorithm Design

Algorithm Design is the process of creating efficient algorithms. There are many approaches to doing so. By using order analysis to compare different algorithms, we can design the one to best fit our needs.

The Searching Problem

The searching problem is as follows:

> Given a list of unique items (assume size = n), find the index of a specific item.

Linear Search Linear Search is the most basic search. The idea is to check every element, returning when we find the one we need.

Example: Where is the element `5` in the array `[1, 2, 3, 4, 5, 6, 7]`

1. At `i = 0`, `arr[0] = 1. 1 != 5.` Continue
2. At `i = 1`, `arr[1] = 2. 2 != 5.` Continue
3. At `i = 2`, `arr[2] = 3. 3 != 5.` Continue
4. At `i = 3`, `arr[3] = 4. 4 != 5.` Continue
5. At `i = 4`, `arr[4] = 5. 5 == 5.`

We found the element at index `4`.

```c
int search(int *arr, int target, int n)
{
    for (int i = 0; i < n; i++)
        if (arr[i] == target)
            return i;
    return -1; // DNE
}
```

This approach works fine, but with certain constraints it proves inefficient.

Binary Search Binary Search is the idea of starting from the middle, and using that value to determine what to do next. If you found target, return the index. If it is smaller than the target, do binary search on the left half. If it is bigger than the target, do binary search on the right half.

This approach is much like searching a dictionary. You wouldn't search each word one by one, you would check a random page and see if you went too far. Only the page would be half way through, and not the middle.

Example: Where is the element `5` in the array `[1, 2, 3, 4, 5, 6, 7]` using Binary Search?

1. Set `low = 0` and `high = 6`

2. Calculate `mid = (low + high) // 2`, so `mid = 3` (index of element `4`). At `mid = 3`, `arr[3] = 4. 4 != 5.` Since `5 > 4`, adjust `low = mid + 1 = 4`.

3. Recalculate `mid = (4 + 6) // 2`, so `mid = 5` (index of element `6`). At `mid = 5`, `arr[5] = 6. 6 != 5.` Since `5 < 6`, adjust `high = mid - 1 = 4`.

4. Recalculate `mid = (4 + 4) // 2`, so `mid = 4` (index of element `5`).

At `mid = 4`, `arr[4] = 5`. `5 == 5`.

We found the element at index `4` and required fewer steps than linear search.

Linear search checks every element of the array. We can represent that as $\sum_{i=0}^{n} 1$, which simplifies to n. Binary search is a bit more complicated to get the closed form solution of (See Recurrence Relations), but because we split the size in half each time, the time complexity is $\log(n)$.

For a small input size, binary search might not be much more efficient, but as n grows larger, binary search becomes much more efficient.

The Sorted Arrays Matching Problem

Given 2 **sorted** arrays, determine all of the values found in both arrays.

The naive approach would be to check every element in the first array against every element in the second array. In other words, $O(n * m)$ where n is the length of one array, and m is the length of another array. This grows far too fast with the size of the input.

A more efficient approeach would be to check if every element in the first array is in the second array using binary search. This is much better with a time complexity of $n\log(m)$. However, we can still do better.

The Linear Approeach Because these arrays are sorted, we can iterate through both of them at the same time. Let i be the index of the first array and j be the index of the second array. If `array1[i] == array2[j]` then we know they match, so we can incremet our counter and both index variables. Otherwise, we can just increment the lower one. We can do this until we reach the end of one of the arrays. The new time complexity is $O(n + m)$

Recursion

Recursion in programming is when a function calls itself.

```c
void is_recursive()
{
    is_recursive();
}
```

The General Formula for Recursion

```
if this is a simple case
    solve it
else
    redefine the problem using recursion
```

Example:

```c
void print_numbers(int low, int high)
{
    // Base case
    if (low >= high)
        return;

    printf("%d\n", low);

    // Recursive call
    print_numbers(low + 1, high);
}
```

Recursion is often better suited to solve problems regarding data structures that are recursive in nature, like linked list, trees and tries (see later sections).

Rerusive solutions tend to have one or more cases with a straight forward solution while the other cases can be reduced to a combination of aforementioned straight forward solutions, or a single one of those cases.

Pros and Cons of Recursion

Pros:

1. Simplfies code
2. Natural fit for some problems
3. Reduces the length of code

Cons:

1. Higher memory overhead than iterative solutions
2. Can be difficult to debug
3. Performance overhead incurred through many recursive calls
4. Recursion is not always intuitive

Types of Recursion

There are two main types of recursion

Direct Recursion In Direct Recursion the function calls itself **directly**.

```c
void print_numbers(int n) {
    if (n == 0)
        return;

    printf("%d ", n);

    print_numbers(n - 1);
}
```

Indirect Recursion In Indirect Recursion the function calls another, which calls itself.

```c
void fn(int n) {
    if (n > 5) return;
    printf("%d ", n);
    fn2(n + 1);
}

void fn2(int n) {
    if (n > 5) return;
    printf("%d ", n);
    fn(n + 1);
}
```

Recurrence Relations

Recurrence relations are equations that describe the time complexity of recursive algorithms in terms of smaller inputs.

Example:

$$T(n) = T(n-1) + c$$

In order to find the time complexity of a recursive function in Big-Oh notation, you must take a recurrence relation and turn it into a closed form solution.

Solving Recurrence Relations using the Iteration Method

Solving recurrence relations can be difficult, but is pretty simple.

The steps are as follows:

1. Iterate T(n) 3 times
2. Solve for the base case = k
3. Substitute in to the kth case

Example 1: $T(n) = T(n-1) + 4$ and $T(0) = 1$

K	T(n)
1	$T(n-1) + 4$
2	$T(n-2) + 4 + 4$
3	$T(n-3) + 4 + 4 + 4$
...	...
k	$T(n-k) + 4k$

Where:

$T(n-1) = T(n-2) + 4$

$T(n-2) = T(n-3) + 4$

And we're looking for the base case, so: $n - k = 0 \rightarrow n = k$

$$T(n) = T(n-k) + 4k$$
$$T(n) = T(n-n) + 4n$$
$$T(n) = T(0) + 4n$$

So,

$$T(n) = 4n + 1$$

In other words, the Big-Oh notation is $O(n)$

Example 2: $T(n) = 2T(n/2) + 6$ and $T(1) = 9$

k	T(n)
1	$2T(n/2) + 6$
2	$2[2T(n/4) + 6] + 6 = 4T(n/4) + 12 + 6$
3	$4[2T(n/8) + 6] + 18 = 8T(n/8) + 24 + 6$
...	...
k	$2^k T(n/2^k) + 6(2^k - 1)$

Now, $n/2^k = 1 \rightarrow k = \log_2 n$.

So we substitute:

$$T(n) = 2^k T(n/2^k) + 6(2^k - 1)$$
$$T(n) = 2^{\log_2 n} T(1) + 6(2^{\log_2 n} - 1)$$
$$T(n) = nT(1) + 6(n - 1)$$
$$T(n) = 9n + 6(n - 1) =$$

So,

$$T(n) = 15n - 6$$

This means that the Big-Oh notation is $O(n)$.

Linked Lists

Linked lists are data types that contain data of any time and a pointer to the next data.

C Implementation:

```
typedef struct node_s
{
    int val;
    node_t *next;
} node_t;
```

Linked lists have a head – the first element in the list, and a tail – the last element in the list.

Example:

```
1 -> 2 -> 3 -> 4
```

Where 1 is the head and 4 is the tail.

Types of Linked Lists

Singly Linked Lists These are the default linked list. They only contain a pointer to the next data point.

Circular Linked Lists Much like their name implies, circular linked lists create a circle where the last node points to the head node.

Example:

```
1 -> 2 -> 3 -> 4 -> 1
```

Doubly Linked Lists Doubly linked lists contain two pointers, a **next** and a **prev**.

```
typedef struct node_s
{
    int val;
    node_t *next;
    node_t *prev;
} node_t;
```

Example:

```
1 <-> 2 <-> 3 <-> 4
```

Circular Doubly Linked Lists Thses lists are no different from regular doubly linked lists, except the tail points back to the head.

Example:

```
4 <- 1 <-> 2 <-> 3 <-> 4 -> 1
```

Next points to the next element in the list. Prev poinst to the previous element in the list.

Pros and Cons

Pros:

1. Dynamic as **fuck**!
2. Simple to modify
3. Can be insrted however (unlike static arrays)

Cons:

1. O(n) Access
2. O(n) Delete
3. O(n) Append*

Note: Append can be O(1) depending on the method and implementation.

Common Implementations of Functions

Create Node

```c
node_t *create_node(int data)
{
    node_t *new_node = (node_t *)malloc(sizeof(node_t));
    if (new_node != NULL)
    {
        new_node->data = data;
        new_node->next = NULL;
    }
    return new_node;
}
```

Is Empty

```c
int is_empty(node_t *head)
{
    return head == NULL;
}
```

Insert (head)

```c
node_t *insert_head(node_t *head, int data)
{
    node_t *new_node = create_node(data);
    if (new_node != NULL)
    {
        new_node->next = head;
    }
    return new_node;
}
```

Insert (tail)

```
node_t *insert_tail(node_t *head, int data)
{
    if (head == NULL)
    {
        return create_node(data);
    }
    head->next = insert_tail(head->next, data);
    return head;
}
```

Delete

```
node_t *delete(node_t *head, int data)
{
    if (head == NULL)
    {
        return NULL;
    }
    if (head->data == data)
    {
        node_t *temp = head;
        head = head->next;
        free(temp);
        return head;
    }
    head->next = delete (head->next, data);
    return head;
}
```

Display

```
void display(node_t *head)
{
    if (head == NULL)
    {
        printf("\n");
        return;
    }
    printf("%d -> ", head->data);
    display(head->next);
}
```

Search

```
node_t *search(node_t *head, int data)
{
    if (head == NULL)
    {
        return NULL;
    }
    if (head->data == data)
    {
        return head;
    }
    return search(head->next, data);
}
```

Free

```
void free_list(node_t *head)
{
    if (head == NULL)
    {
        return;
    }
    free_list(head->next);
    free(head);
}
```

Stacks

Stacks are a data structure that is **Last in, Last Out** or LIFO. This means the last element pushed to the stack is at the top of the stack.

In c specifically, there are multiple ways to implement a stack. My preference would be as such:

```c
#define MAX 100
typedef struct stack_s {
    int items[MAX];
    int top;
} stack_t;
```

where the integer top represents which index the top of the stack is at. An index of `-1` means it is empty. An index of `99`, or `MAX - 1` means it is full.

To Push: increment top and set `items[top] = data` To Retreive: return `items[top]` To Pop: decrement top

Reverse Polish Notation (RPN)

Reverse Polish Notation, or postfix notation is a way of representing mathematical equations. Instead of writing `1 + 1`, you would write `1 1 +`.

Evaluating RPN To evaluate RPN, you do the following algorithm:

1. Scan tokens left to right
2. If the token is a number, push to stack
3. If the token is an operator:
    - Pop two numbers from the stack
    - Apply the operator
    - Push the result back to the stack
4. At the end, the result is on the top of the stack.

Example: `3 4 + 2 *`

Step 1:

```
3 4 + 2 *
|
```

Stack: [3]

Add 3 to stack

Step 2:

```
3 4 + 2 *
  |
```

Stack: [3, 4]

Add 4 to stack

Step 3:

```
3 4 + 2 *
    |
```

Stack: [7]

Pop 3 and 4

Add them (the operator is a `+`)

Push the sum (`7`) to the stack

Step 4:

```
3 4 + 2 *
      |
```

Stack: [7, 2]

Push 2 to the stack

Step 5:

```
3 4 + 2 *
        |
```

Stack: [14]

Pop 7 and 2 from the stack

Multiply them (the operator is a `*`)

Push the product (`14`) to the stack

Code for Stacks

Initiation

```c
void init(stack_t *s)
{
    s->top = -1;
}
```

Is Empty

```c
int is_empty(stack_t *s)
{
    return s->top == -1;
}
```

Is Full

```c
int is_full(stack_t *s)
{
    return s->top == MAX - 1;
}
```

Push

```c
void push(stack_t *s, int val)
{
    if (is_full(s))
    {
        // Stack is full and cannot be added to
        exit(1);
    }
    s->items[++s->top] = val;
}
```

Pop

```c
int pop(stack_t *s)
{
    if (is_empty(s))
    {
        // Stack is empty and cannot be popped from
        exit(1);
    }
    return s->items[s->top--];
}
```

Stack Questions

Easy: Valid Parentheses (#20); Min Stack (#155); Implement Stack using Queues (#225)

Medium: Evaluate Reverse Polish Notation (#150); Next Greater Element I (#496); Next Greater Element II (#503); Daily Temperatures (#739); Asteroid Collision (#735); Binary Tree Inorder Traversal (#94)

Hard: Largest Rectangle in Histogram (#84); Basic Calculator (#224); Basic Calculator II (#227); Basic Calculator III (#772)

Queues

Queues are the opposite of stacks – they follow a **First in, First Out**, or FIFI, pattern. That is, the first element added to the queue is the first one to be removed.

The queue structure would be as follows:

```
#define MAX 100

typedef struct {
    int items[MAX];
    int front, rear, size;
} queue_t;
```

The integer front points to the front element of the queue. Rear points to the last element of the queue, and size contains the number of elements in the queue.

Code for Queues

Initiation

```
void init(queue_t *q)
{
    q->front = 0;
    q->rear = -1;
    q->size = 0;
}
```

Is Empty

```
int is_empty(queue_t *q)
{
    return q->size == 0;
}
```

Is Full

```
int is_full(queue_t *q)
{
    return q->size == MAX;
}
```

Enqueue

```
void enqueue(queue_t *q, int val)
{
    if (is_full(q))
    {
        // Queue is full and cannot be added to
        exit(1);
    }
    q->rear = (q->rear + 1) % MAX;
    q->items[q->rear] = val;
    q->size++;
}
```

Dequeue

```
int dequeue(queue_t *q)
{
    if (is_empty(q))
    {
        // Queue is empty and cannot be dequeued from
        exit(1);
    }
    int val = q->items[q->front];
    q->front = (q->front + 1) % MAX;
    q->size--;
    return val;
}
```

Queue Questions

Easy: Implement Queue using Stacks (#232); Number of Recent Calls (#933); Moving Average from Data Stream (#346)

Medium: Rotting Oranges (#994); Course Schedule (#207); Perfect Squares (#279); Open the Lock (#752); Dota2 Senate (#649)

Hard: Sliding Window Maximum (#239); LFU Cache (#460); The Maze II (#505)

Binary Tree

A Binary Tree is a data structure similar to Linked List, except that it has two children instead of one. There is a left child and a right child.

Terminology:

Node: The unit of a binary tree, holding some data element Root: The topmost node in the tree. It has no parent Leaf: A node that has no children Parent: A node thas has children Child: A node that is a descendant (left or right) of another node Subtree: A tree formed from a node and all its descendants Depth: The number of edges (connections)from a root to the node Height: The number of edges on the longest path from the node to a leaf

Types of Binary Trees:

Full Binary Tree: Every node either has 0 or 2 children Complete Binary Tree: Every level is completely filled except possibly for the last level Perfect Binary Tree: All internal nodes have 2 children, and all leaf nodes are at the same level

Traversing a Binary Tree

There are many ways to traverse a binary tree, or to visit all of the nodes in the tree.

In-order In-order, or left-node-right traverses the left subtree, the node is visited, then the right subtree is traversed.

Pre-order Pre-order is node-left-right, where the node is visited first, and then the left and right are traversed.

Post-order Post-order is left-right-node, where the node is visited last.

Level-order (Breadth-First) Level-order is more complicated than the other three. This traversal goes level by level, right to left.

Starting with the root node in the queue, the top of the queue is popped, the left and right chiidren are added to the queue, and computation (or printing) is done on the last.

Code:

```
void bfs(node_t *root)
{
    if (root == NULL)
    {
        return;
    }

    queue_t q;
    init_queue(&q);
    enqueue(&q, root);
    while (!is_empty(&q))
    {
        node_t *current = dequeue(&q);
        printf("%d ", current->data);

        if (current->left)
        {
            enqueue(&q, current->left);
        }
        if (current->right)
        {
            enqueue(&q, current->right);
        }
    }
}
```

Binary Tree Questions

Easy: Maximum Depth of Binary Tree (#104); Invert Binary Tree (#226); Symmetric Tree (#101); Path Sum (#112); Diameter of Binary Tree (#543)

Medium:
Binary Tree Level Order Traversal (#102); Construct Binary Tree from Inorder and Postorder Traversal (#106); Binary Tree Right Side View (#199); Binary Tree Zigzag Level Order Traversal (#103); Sum Root to Leaf Numbers (#129)

Hard: Binary Tree Maximum Path Sum (#124); Recover Binary Tree (#99); Serialize and Deserialize Binary Tree (#297); All Nodes Distance K in Binary Tree (#863)

Binary Search Trees (BSTs)

Binary Search Trees are special binary trees. In a binary search tree, the left child is less than the root, and the right child is greater than the root.

Example:

```
       8
     /   \
    3     10
   / \      \
  1   6      14
     / \    /
    4   7  13
```

node	check
8	$3 < 8 < 10$
3	$1 < 3 < 6$
6	$4 < 6 < 7$
10	$10 < 14$
14	$13 < 14$

BST Operations

The main BST operations are Search, Insert, Delete, and Traverse

Search A BST search involves repeatedly searching subtrees, not too dissimlar to a binary search.

Time Complexity:

- Worst: $O(n)$ – The entire tree could consist of only right or left children requiring that the entire tree be searched.
- Best: $O(1)$
- Average: $O(\log_2 n)$

Insert A BST Insert adds a new value to the BST. In order to do this, you must recursively go down the tree. Go left if the value is less than the node, and right if the value is greater than the node.

Code:

```
node_t* insert(node_t* root, int val)
{
    if (root == NULL) return create_node(val);
    if (val < root->data)
        root->left = insert(root->left, val);
    else if (val > root->data)
```

```
        root->right = insert(root->right, val);
    return root;
}
```

Delete A Deletion from a BST is slightly more complicated. First, the value must be found. Then, it must be replaed with the inorder successor (which is the min of the right subtree).

Code:

```
node_t* find_min(node_t* root) {
    while (root->left != NULL) root = root->left;
    return root;
}

node_t* delete(node_t* root, int val) {
    if (root == NULL) return NULL;
    if (val < root->data)
        root->left = delete(root->left, val);
    else if (val > root->data)
        root->right = delete(root->right, val);
    else {
        // One child or no child
        if (root->left == NULL) {
            node_t* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            node_t* temp = root->left;
            free(root);
            return temp;
        }
        // Two children: get inorder successor (min in right subtree)
        node_t* temp = find_min(root->right);
        root->data = temp->data;
        root->right = delete(root->right, temp->data);
    }
    return root;
}
```

Traverse Traversal of the BST is the same as the traversal of a regular Binary Tree except the in-order traversal of a BST is sorted. This is because the left is less than the node, and the right is greater than the node.

Binary Search Tree (BST) Questions

Easy:
Search in a Binary Search Tree (#700); Validate Binary Search Tree (#98); Minimum Absolute

Difference in BST (#530); Convert Sorted Array to BST (#108)

Medium:
Delete Node in a BST (#450); Lowest Common Ancestor of a BST (#235); Trim a Binary Search Tree (#669); Convert BST to Greater Tree (#538); Closest Binary Search Tree Value (#270)

Hard:
Recover Binary Search Tree (#99); Count of Smaller Numbers After Self (#315); Maximum Sum BST in Binary Tree (#1373)

AVL Trees

An AVL Tree is a self-balancing binary search tree. It maintains a balance factor – the height difference between left and right subtrees – for each node,ensures the balance factor is between -1 and 1.

If an insertion or deletion causes an unbalance, it must be rotated.

TODO: AVL Tree rotations and examples

Tries

Tries are special trees that contain letters. Each node can have 26 children, one for each letter. Words are represented as a path from the root to a node flagged as a word.

Struct Representation:

```c
#define ALPHABET_SIZE 26

typedef struct trie_s
{
    struct trie_s *children[ALPHABET_SIZE];
    int flag;
} trie_t;
```

Operations

There are 4 main operations.

1. Insert - Add a word to the dictionary/list
2. Search - Check if a word has been added
3. Starts With - Check if any words start with a given prefix
4. Delete - Remove a given word

Insert Insert adds a word to the list. We do this by iterating down each character in the word. If it is there, we just increment the letter. If it is not, we add the node, and then increment the letter.

Time Complexity: $O(n)$, where n is the length of the word

C Implementation:

```c
void insert(trie_t* root, const char* word)
{
    trie_t* current = root;
    for (int i = 0; word[i] != '\0'; i++)
    {
        int index = word[i] - 'a';
        if (current->children[index] == NULL)
            current->children[index] = createNode();
        current = current->children[index];
    }
    current->flag = 1;
}
```

Search Search checks if a word is in the list. We can iterate down each character – like in insert – and return 1 if we reach the final character and flag is true. If at any point a node doesn't exist, we can return 0, or if the flag is not true we can return 0.

Time Complexity: $O(n)$, where n is the length of the word

C Implementation:

```
int search(trie_t* root, const char* word)
{
    trie_t* current = root;
    for (int i = 0; word[i] != '\0'; i++)
    {
        int index = word[i] - 'a';
        if (current->children[index] == NULL)
            return 0;
        current = current->children[index];
    }
    return current->flag;
}
```

Starts With To check if any word starts with a given prefix, we can traverse down the trie, and return true if the last character is reached.

Time Complexity: $O(n)$, where n is the length of the prefix

C Implementation:

```
bool starts_with(trie_t* root, const char* prefix)
{
    trie_t* current = root;
    for (int i = 0; prefix[i] != '\0'; i++)
    {
        int index = prefix[i] - 'a';
        if (current->children[index] == NULL)
            return false;
        current = current->children[index];
    }
    return 1;
}
```

Delete Delete removes a word from the trie, and clears the remaining empty nodes. In order to do this we recursively delete the children of each character, and then remove the node if it is empty.

Time Complexity: $O(n)$, where n is the length of the word

C Implementation:

```
trie_t* delete_word(trie_t* root, const char* word, int depth)
{
    if (!root) return NULL;

    if (word[depth] == '\0')
    {
        root->flag = 0;

        if (is_empty(root))
        {
            free(root);
            return NULL;
        }
        return root;
    }

    int index = word[depth] - 'a';
    root->children[index] = delete_word(root->children[index], word, depth + 1);

    if (is_empty(root) && root->flag == 0)
    {
        free(root);
        return NULL;
    }

    return root;
}
```

Heaps

A heap (separate from the heap in memory) is a form of a tree – represented internally as an array – that satisfies the heap property. The heap property is the rule that every parent node is **less than or equal/greater than or equal** to its children for min-heaps and max-heaps respectively.

Heaps are complete binary trees (see Binary Tree unit). This means that all levels of the tree are filled except the last, and it is filled from left to right.

Array Representation

Heaps are represented internally as an array instead of the traditional tree data structure. Node i has children at index $2i + 1$ and $2i + 2$.

Example:

```
         3
       /   \
      11    5
     / \   /
   10  7  6
```

Array: [3, 11, 5, 10, 7, 6]

Node 1 (11) has children at index 3 (10) and 4 (11).

Why?

Heaps are an implementation of a priority queue, where the smallest/largest element can be accessed efficiently.

Time Complexity:

- Insert: $O(\log n)$
- Extract max/min: $O(\log n)$
- Peek max/min: $O(1)$

Creating Heaps

Heapify The heapify function swaps around the values of the current node and its children so that it satisfies the heap property. If one of the children gets swapped with the parent, the heapify function must also be called on the child that was swapped so that it continues to satisfy the heap property.

Example:

Before heapify call on 3:

```
         3
       /   \
      11    5
     / \   /
   10  7  6
```

After first heapify:

```
       11
      /  \
     3    5
    / \    /
  10   7  6
```

Notice how 3 doesn't satisfy the max-heap property.

After second heapify:

```
       11
      /  \
     10   5
    / \    /
   3   7 6
```

Building a Heap The main way discussed in CS1 to build a heap is the bottom up method. This means to perform the heapify function from an unsorted array from the last non-lead node up to the root.

Example: Build a min-heap with the unsorted array [9, 4, 7, 1, -2, 6, 5]

0. [9, 4, 7, 1, -2, 6, 5]
1. [9, 4, 5, 1, -2, 6, 7] - Heapify node 2 (7). Swap 5 and 7
2. [9, -2, 5, 1, 4, 6, 7] - Heapify node 1 (4). Swap 4 and -2
3. [-2, 9, 5, 1, 4, 6, 7] - Heapify node 0 (9). Swap 9 and -2. 9 no longer satisfies heap property.
4. [-2, 1, 5, 9, 4, 6, 7] - Heapify node 1 (9). Swap 9 and 1

Heap Sort

Heap sort is a sorting algorithm (see sorting algorithms). It builds a heap and then repeatedly swaps the root (min/max) with the last element and reducing the heap size, maintaining the heap property each time.

Heap sort is in-pace (no extra memory) and is always $O(n \log n)$.

Example: [4, 2, 1, 7, 3, 8]

0. [4, 2, 1, 7, 3, 8]
1. **Build Max-Heap**:
   - Heapify from last non-leaf $\rightarrow$ first
   - After heapify: [8, 4, 1, 2, 3, 7]
2. **Sort**:
   - Swap root with last: [7, 4, 1, 2, 3, 8] Heapify: [4, 3, 1, 2, 7, 8]
   - Swap: [2, 3, 1, 4, 7, 8] Heapify: [3, 2, 1, 4, 7, 8]
   - Swap: [1, 2, 3, 4, 7, 8] Heapify: [2, 1, 3, 4, 7, 8]
   - Keep going until fully sorted: [1, 2, 3, 4, 7, 8]

Heap Code

```
void heapify(int arr[], int n, int i)
{
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] < arr[smallest])
        smallest = left;

    if (right < n && arr[right] < arr[smallest])
        smallest = right;

    if (smallest != i)
    {
        swap(&arr[i], &arr[smallest]);
        heapify(arr, n, smallest);
    }
}

void build_heap(int arr[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
    {
        heapify(arr, n, i);
    }
}
```

Hash Tables

Hash Tables are a key-value store of data. Much like how an array stores data with integer keys, hash tables store data with, in theory, any data type as a key.

They way they do this is with a hash function to translate your given data type into an intger key for an underlying array. Typically this also means modding a previously computed value by the size of the array.

Example: Insert ("apple", 7) with the hash function h(x) that returns the sum of all characters c mod 10 (the size of the array).

Hash "apple" $\rightarrow 97 + 112 + 112 + 108 + 101 = 530 \% 10 = 0$

Array: [("apple", 7), X, X, X, X, X, X, X, X, X]

Collisions

Collisions are when a hash function produces duplicates values. For example, using the above hash function for "apple" and for "appkf" produce the same index. There are a few methods to prevent collisions.

Chaining Chaining solves this issue by saving each value in the underlying array as a linked list and, thus, chains overlap.

Example: Insert ("apple", 7) and ("appkf", 10) with the hash function h(x) that returns the sum of all characters c mod 10.

Array: [("apple", 7) -> ("appkf", 10), X, X, . . .]

Pros:

- Simple to implement
- Handles collisions easily
- Tables can grow father than their underlying array's size

Cons:

- Slower lookup (worst case $O(n)$)
- Uses more memory than other methods

Linear Probing Linear probing solves this issue in a different way. By incrementing the hash until there is an open slot. To prevent overflow, the hash must once again be modded by the value of the array.

Example: Insert ("apple", 7), ("appkf", 10), and ("appjf", 13) with the hash function h(x) that returns the sum of all characters c mod 10.

Array: [("apple", 7), ("appkf", 10), ("appjf", 13), X, X, . . .]

Pros:

- No extra memory
- Fast with a low load factor

Cons:

- Deletion is complicated
- Bad performance with a high load factor

Quadratic Probing Quadratic probing is very similar to linear probing, except instead of incrementing by 1, we increment in squares.

Example: Insert ("apple", 7), ("appkf", 10), and ("appjf", 13) with the hash function h(x) that returns the sum of all characters c mod 10.

Array: [("apple", 7), ("appkf", 10), X, ("appjf", 13), X, X, ...]

Pros:

- Better performance with higher load

Cons:

- Still relatively complicated for deletion
- More complex than linear probing

Time Complexity

The time complexity of inserting and retreiving is generally close to $O(1)$. With a large load or a bad hash function, the time complexity can approach $O(n)$ where n is the number of elements.

Sorting Algorithms

Sorting algorithms are algorithms that solve the sorting problem. That is, more formally, given n unsorted elements, rearrange them in a way such that $x_0 < x_1 < ... < x_n$.

There are two main types of algorithms discussed in CS1. Algorithms that are $O(n^2)$, and algorithms that are $O(n \log n)$.

Selection Sort

Selection sort is, in my opinion, the most naive sorting algorithm. Selection sort **selects** the minimum element each iteration, and swaps it to the correct spot.

Example: [4, 2, 1, 7, 3, 8]

0. [4, 2, 1, 7, 3, 8]
1. [1, 2, 4, 7, 3, 8] - Select 1, swap it with i=0, 4
2. [1, 2, 4, 7, 3, 8] - Select 2, swap it with i=1, 2 (no change)
3. [1, 2, 3, 7, 4, 8] - Select 3, swap it with i=2, 4
4. [1, 2, 3, 4, 7, 8] - Select 4, swap it with i=3, 7
5. [1, 2, 3, 4, 7, 8] - Select 7, swap it with i=4, 7 (no change)
6. [1, 2, 3, 4, 7, 8] - Select 8, swap it with i=5, 8 (no change)

***Time Complexity**:

- Worst: $O(n^2)$
- Best: $O(n^2)$
- Average: $O(n^2)$

Tip: remember that selection sort **selects** the minimum each iteration

Bubble Sort

Bubble sort is named because the largest element **bubbles** up to the correct position with each iteration. During each pass through the array, adjacent elements are compared, and if they are out of order, they are swapped. This process is repeated until the entire array is sorted.

Example: [4, 2, 1, 7, 3, 8]

0. [4, 2, 1, 7, 3, 8]
1. [2, 1, 4, 3, 7, 8] - 4 swaps with 2; 4 swaps with 1; 7 swaps with 3.
2. [1, 2, 3, 4, 7, 8] - 2 swaps with 1; 4 swaps with 3

***Time Complexity**:

- Worst: $O(n^2)$
- Best: $O(n)$
- Average: $O(n^2)$

Note: Unlike selection sort, Bubble Sort can be O(n) best case if the array is already sorted.

Tip: remember that in bubble sort, the larger elements **bubble** up.

Insertion Sort

Insertion sort is similar to both selection sort and bubble sort. Each iteration in insertion sort we add the next element to its correct spot in the new array.

Example: [4, 2, 1, 7, 3, 8]

0. [**4**, 2, 1, 7, 3, 8]
1. [**2**, **4**, 1, 7, 3, 8] - 2 is inserted in front of 4
2. [**1**, **2**, **4**, 7, 3, 8] - 1 is inserted in front of 2
3. [**1**, **2**, **4**, **7**, 3, 8] - 7 is inserted behind 4
4. [**1**, **2**, **3**, **4**, **7**, 8] - 3 is inserted between 2 and 4
5. [**1**, **2**, **3**, **4**, **7**, **8**] - 8 is inserted after 7

Tip: remember that insertion sort **inserts** the next element to its correct spot.

***Time Complexity**:

- Worst: $O(n^2)$
- Best: $O(n)$
- Average: $O(n^2)$

Merge Sort

Merge sort is the most simple $O(n \log n)$ sorting algorithm. Merge sort divides the array into 2 portions, sorts them, and merges them.

The drawback with merge sort is that it uses $O(n \log n)$ space complexity, so it is very inefficient in that respect.

Example: [4, 2, 1, 7, 3, 8]

0. [4, 2, 1, 7, 3, 8]
1. [4, 2, 1] [7, 3, 8] - call 1
2. [4, 2] [1] [7] [3, 8] - call 2
3. [2, 4] [1] [7] [3, 8] - sort
4. [1, 2, 4] [3, 7, 8] - merge 1
5. [1, 2, 3, 4, 7, 8] - merge 2

Time Complexity: $O(n \log n)$ **Space Complexity**: $O(n \log n)$

Quick Sort

Quick sort is a divide-and-conquer sorting algorithm. It selects a pivot, partitioning the lesser elements to the left, and the greater elements to the right. Then the partitions are merged.

The key advantage of quick sort is that it is in-place (requires very little extra space), unlike merge sort.

Example: `[4, 2, 1, 7, 3, 8]`

0. [4, 2, 1, 7, 3, 8]
1. Choose pivot `8` $\rightarrow$ Partition: [4, 2, 1, 7, 3] [8] []

2. Sort [4, 2, 1, 7, 3]:
   - Pivot `3` → [2, 1] [3] [4, 7]
   - Sort [2, 1]: [1, 2]
   - Sort [4, 7]: [4, 7]
   - Merge: [1, 2] + [3] + [4, 7] → [1, 2, 3, 4, 7]
3. Merge: [1, 2, 3, 4, 7] + [8] + [] → [1, 2, 3, 4, 7, 8]

Time Complexity:

- **Best / Average**: $O(n \log n)$

- **Worst Case**: $O(n^2)$ (when pivot is always smallest/largest element)

Base Conversion

In base 10, or the decimal system that we use for numbers, any number can be represented as $d_0 * 10^0 + d_1 * 10^1 + d_2 * 10^2 + ... + d_n * 10^n$

Example: $1029 = 9 * 10 + 2 * 10^2 + 0 * 10^3 + 1 * 10^3$

Base 2 to Base 10

We can do the exact same thing in binary, except we use 2 because it is base 2. Where a binary number can be converted to decimal like $d_0 * 2^0 + d_1 * 2^1 + d_2 * 2^2 + ... + d_n * 2^n$

Example: Convert 00100111_2 to decimal.

$1 * 2^0 + 1 * 2^1 + 1 * 2^2 + 0 * 2^3 + 0 * 2^4 + 1 * 2^5 + 0 * 2^6 + 0 * 2^7 = 39$

Base 2 to Base 16

Converting base 2 to base 16, or hex, is trivial. 4 bits can represent 16 different numbers. As such, each 4 block of a binary digit represents a single base 16 digit (which is represented as 0-F).

Conversion Chart:

base 2	base 16
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Note: This conversion chart is super easy to figure out. Do not memorize it.

Example: Convert 00100111_2 to base 16.

$0010 \rightarrow 2$ and $0111 \rightarrow 7$ so 27

Base 2 to Base 4 and 8

THe same trick applies to base 4 and base 8, only the number of bits is smaller.

Decimal to Base 2

To convert a decimal number to binary, you repeatidly divide the number by 2 and keep track of the remainders.

Steps:

1. Divide the number by 2.
2. Write down the remainder (either 0 or 1).
3. Update the number to be the quotient of the division.
4. Repeat until the number becomes 0.
5. The binary number is the remainders read in reverse order (bottom to top).

Example: Convert 13_10 to base 2.

```
13 ÷ 2 = 6, remainder = 1
 6 ÷ 2 = 3, remainder = 0
 3 ÷ 2 = 1, remainder = 1
 1 ÷ 2 = 0, remainder = 1
```

So, 13_10 in binary is 1101_2.

Decimal to Any Other Base

To convert decimal to any other base, follow the same process but divide by the desired base instead of 2.

Example: Convert 26_10 to base 3.

```
26 ÷ 3 = 8 remainder 2
 8 ÷ 3 = 2 remainder 2
 2 ÷ 3 = 0 remainder 2
```

So, 26_10 to base 3 is 222_3.

2's Complement

Representing negative numbers isn't trivial in binary. There are 3 main methods: sign magnitude, 1's complement, and 2's complement.

Sign Magnitude

Sign magnitude is a method of storing the sign of the number in the MSB of the number.

Example:

$10100011_2 = -35_10$

$00100011_2 = 35_10$

Pros:

- Sign magnitude is very simple and easy to understand.

Cons:

- 0 is represented twice. Once as 0 and once as -0
- Arithmetic is very difficult

1's Complement

1's Complement is a way of representing a negative number as the inverse of a positive number. That is, more formally, that $x_2 = -x_10$, or that the inverse of a number in base 2 is the negative of the number in base 10.

Example:

$00100011_2 = 35_10$

$11011100_2 = -35_10$

Pros:

- The representation of 0 and -0 is easier to deal with than in sign magnitude
- Arithmetic is much easier

Cons:

- There are still two representations of 0
- Arithmetic is still not straight forward

2's Complement

2's Complement is the modern approach to representing numbers in binary.

Negating Numbers To negate binary numbers we:

1. Invert the bits
2. Add 1

Example: What is -35_10 in 2's complement.

0. $35_10 = 00101100_2$
1. $00101100_2 \rightarrow 11011100_2$
2. $00101100_2 + 1_2 = 11011101_2$

So, -35_10 is 11011101_2 in 2's complement binary.

Converting 2's Complement to base 10 To revert the process:

1. Check the number is actually negative
2. Invert the bits
3. Add 1
4. Convert to binary

Example: Convert 2's complement 11000111_2 to decimal.

1. MSB is 1, so we continue
2. $11000111_2 \rightarrow 00111000_2$
3. $00111000_2 + 1_2 = 00111001_2$
4. $00111001_2 = 57$

So, 11000111_2 is -57.

Bitwise Operators

Bitwise Operators are how we manipulate binary numbers. There are a few main bitwise operators: `AMD`, `OR`, `XOR`, and `NOT`.

These operators apply the base logical operator to each bit.

AND

The AND operator applies AND to each bit. This means that each bit is only on if both bits of the inputs are on.

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

So, a binary A and B would be $A_0 \& B_0$, $A_1 \& B_1$, $\ldots$, for every corresponding bit in the two numbers.

Example: Compute 1101 **AND** 1011

1101 **AND** 1011 = 1111

OR

The OR operator applies OR to each bit.

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

So, a binary A or B would be $A_0|B_0$, $A_1|B_1$, $\ldots$, for every corresponding bit in the two numbers.

Example: Compute 1101 **OR** 1011

1101 **OR** 1011 = 1111

XOR

The XOR (exclusive OR) operator applies XOR to each bit. This means one or the other, but not both.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

So, a binary A XOR B would be $A_0 \oplus B_0$, $A_1 \oplus B_1$, $\ldots$, for every corresponding bit in the two numbers.

Example: Compute 1101 **XOR** 1011

1101 **XOR** 1011 = 0110

NOT

The NOT operator inverts each bit of the number (also called the bitwise negation).

A	NOT A
0	1
1	0

Example: Compute **NOT** 1101

NOT 1101 = 0010

Bit Masking

Bit masking is a technique to manipulate bits of a binary number. We use bitwise operators (see above) with a "mask". A mask is a set of bits that indicate what should be modified.

Numbers can be thought as sets of bits. Manipulating these bits through bitwise shifts and comparison operators provides the bits we are looking for.

Tip: In C, we can use the format $0bXXXX$, where X is either a one or zero, to write binary numbers.

Example: Check if a number n has bits 5, 6, and 7 set.

We can do $(n >> 5)\ \&\ 0b111 == 0b111$.

Types of Bit Masks

1. Setting to 1 - Use the `OR` operator `|` with a 1
2. Setting to 0 - Use the `AND` operator `&` with a 0
3. Toggling bits - Use the `XOR` to flip or invert specific bits
4. Checking bits - Use the `AND` operator to check specific bits

Example: Student Information

A student is stored as an 8 bit number where:

- **Bit 1**: Student has a library card (1 if true, 0 if false)
- **Bit 2**: Student is enrolled in a course (1 if true, 0 if false)
- **Bit 3**: Student has completed an internship (1 if true, 0 if false)
- **Bit 4**: Student is a graduate (1 if true, 0 if false)
- **Bit 5**: Student is a part-time student (1 if true, 0 if false)
- **Bit 6**: Student has a scholarship (1 if true, 0 if false)
- **Bit 7**: Student is a member of a club (1 if true, 0 if false)
- **Bit 8**: Student is in good academic standing (1 if true, 0 if false)

Questions:

1. Write a function to check if a student has a library card.
2. Write a function to check if a student is an A tier student (has a scholarship, is in good academic standing, and is enrolled in a course)

Answers:

1.

```c
int has_library_card(unsigned char student_info)
{
    // Check if bit 1 (least significant bit) is set to 1
    return (student_info & (1 << 0)) != 0;
}
```

2.

```c
int is_a_tier_student(unsigned char student_info)
{
    // Check if the student is enrolled in a course (bit 2), has a scholarship (bit 6), and in good
    return (student_info & (1 << 1)) &&  // Check if bit 2 (enrolled) is 1
           (student_info & (1 << 5)) &&  // Check if bit 6 (scholarship) is 1
           (student_info & (1 << 7));    // Check if bit 8 (good academic standing) is 1
}
```