Notes for ECE 36800 - Data Structures

Zeke Ulrich

March 11, 2025

Contents

Course Description

Provides insight into the use of data structures. Topics include stacks, queues and lists, trees, graphs, sorting, searching, and hashing.

Introduction

Algorithms

An algorithm is simply a method to solve a class of problems. There are algorithms to make toast, to solve a Rubik's cube, and to plot the optimal path for a mailmain. The two most important qualities of any algorithm are effectiveness and secondly efficiency. Our solution must work, preferably in as little time as possible. A working algorithm is no good if the heat death of the universe occurs before it finds a solution. We describe the cost of an algorithm using Big O notation. As an example, consider listing 1 If each operation has cost C_i, then the total

```
int total = 0; // C_1
for (int i = 0; i++; i <= n){ // C_2
    total = total + i; // C_3
}
return total; // C_4
```

Figure 1: Sum of first n numbers

cost of the program is

$$C_1 + C_2(n+1) + C_3 n + C_4 = n(C_2 + C_3) + (C_1 + C_4). \qquad (1)$$

Big O notation considers only the largest power of n, so the Big O complexity for this algorithm would be $O(n)$.

Data Structures

A data structure is an organization of information for ease of manipulation. For example, a dictionary, a checkout line, and an org chart.

Sorting

Bubble Sort

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

Time Complexity:

- Best Case: $O(n)$ (when the array is already sorted)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

Listing 1: Bubble Sort

```
void bubble_sort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        int swapped = 0;
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                swapped = 1;
            }
        }
        if (swapped == 0)
            break;
    }
}
```

Insertion Sort

Insertion Sort builds the sorted array one item at a time. It picks an element and places it into its correct position in the sorted part of the array.

Time Complexity:

- Best Case: $O(n)$ (when the array is already sorted)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

Listing 2: Insertion Sort

```
void insertion_sort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Selection Sort

Selection Sort divides the array into a sorted and an unsorted region. It repeatedly selects the smallest (or largest) element from the unsorted region and moves it to the sorted region.

Time Complexity:

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

Listing 3: Selection Sort

```
void selection_sort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_idx = i;
        for (int j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx])
                min_idx = j;
        }
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Shell Sort

Shell Sort is an optimization over Insertion Sort. It first sorts elements that are far apart and then progressively reduces the gap between elements to be sorted.

Time Complexity:

- Best Case: $O(n \log n)$
- Average Case: depends on the gap sequence, commonly $O(n^{3/2})$ or $O(n \log^2 n)$
- Worst Case: $O(n^2)$

Listing 4: Shell Sort

```
void shell_sort(int arr[], int n) {
    for (int gap = n/2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}
```

A sorting algorithm is stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted. Insertion sort, merge sort, and bubble sort are all stable sorting algorithms.

Asymptotic Notation

Big O

Big O notation describes the upper bound of an algorithm's growth rate. It provides an asymptotic measure of the time complexity or space complexity of an algorithm as a function of the input size. Informally, big O notation just picks out the highest order term from your runtime expression. Formally, Big O is defined as follows:

$$f(n) \in O(g(n)) \iff \exists\, c > 0 \text{ and } n_0 \geq 0 \text{ such that } \forall n \geq n_0,\ 0 \leq f(n) \leq c \cdot g(n).$$

- $f(n)$: The function representing the actual growth rate of the algorithm.
- $g(n)$: The comparison function, often representing the dominant term of $f(n)$.
- $c > 0$: A constant that scales $g(n)$ to bound $f(n)$.
- n_0: A constant representing the threshold beyond which the inequality holds.

Big O notation focuses on the dominant term of $g(n)$, ignoring lower-order terms and constant factors, to provide a simplified representation of the algorithm's efficiency.

Big Ω

Big Ω notation describes the lower bound of an algorithm's growth rate. It provides an asymptotic measure of the minimum time complexity or space complexity of an algorithm as a function of the input size. While Big O focuses on the upper bound, Big Ω focuses on the lower bound, ensuring that the algorithm's growth rate is at least as fast as a specified function, up to constant factors. Formally, Big Ω is defined as follows:

$$f(n) \in \Omega(g(n)) \iff \exists\, c > 0 \text{ and } n_0 \geq 0 \text{ such that } \forall n \geq n_0,\ 0 \leq c \cdot g(n) \leq f(n).$$

- $f(n)$: The function representing the actual growth rate of the algorithm.
- $g(n)$: The comparison function, representing the lower bound on $f(n)$.
- $c > 0$: A constant that scales $g(n)$ to bound $f(n)$ from below.
- n_0: A constant representing the threshold beyond which the inequality holds.

Big Ω notation is used to describe the best-case or minimum growth rate of an algorithm. For example, if an algorithm's runtime is $\Omega(n \log n)$, it means that the algorithm cannot perform better than $n \log n$ in the best case, up to constant factors. This notation is particularly useful when proving lower bounds on the complexity of problems or algorithms.

Big Θ

Big Θ notation describes both the upper and lower bounds of an algorithm's growth rate. It provides a tight asymptotic bound on the time complexity or space complexity of an algorithm as a function of the input size. Unlike Big O, which only provides an upper bound, Big Θ ensures that the growth rate of the algorithm is bounded both above and below by the same function, up to constant factors. Formally, Big Θ is defined as follows:

$$f(n) \in \Theta(g(n)) \iff \exists\, c_1, c_2 > 0 \text{ and } n_0 \geq 0 \text{ such that } \forall n \geq n_0,\ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n).$$

- $f(n)$: The function representing the actual growth rate of the algorithm.
- $g(n)$: The comparison function, representing the tight bound on $f(n)$.
- $c_1, c_2 > 0$: Constants that scale $g(n)$ to bound $f(n)$ from below and above.
- n_0: A constant representing the threshold beyond which the inequality holds.

Big Θ notation is used when the growth rate of an algorithm is known to be tightly constrained by a specific function, providing a precise characterization of its efficiency. For example, if an algorithm's runtime is both $O(n^2)$ and $\Omega(n^2)$, then its runtime is $\Theta(n^2)$.

Little o

Little o notation describes an upper bound that is not tight. It provides a stricter asymptotic measure of the growth rate of an algorithm compared to Big O notation. While Big O allows for the possibility that the growth rates are the same up to constant factors, Little o ensures that the growth rate of the function is strictly less than the comparison function. Formally, Little o is defined as follows:

$$f(n) \in o(g(n)) \iff \forall\, c > 0, \exists\, n_0 \geq 0 \text{ such that } \forall n \geq n_0,\ 0 \leq f(n) < c \cdot g(n).$$

- $f(n)$: The function representing the actual growth rate of the algorithm.

- $g(n)$: The comparison function, representing the upper bound on $f(n)$.
- $c > 0$: A constant that scales $g(n)$ to bound $f(n)$ strictly from above.
- n_0: A constant representing the threshold beyond which the inequality holds.

Little o notation is used to describe situations where the growth rate of an algorithm is strictly smaller than the comparison function. For example, if $f(n) = 2n + 3$, then $f(n) \in o(n^2)$, because $2n + 3$ grows strictly slower than n^2 as n becomes large. This notation is particularly useful when emphasizing that an algorithm's growth rate is asymptotically insignificant compared to another function.

Little ω

Little ω notation describes a lower bound that is not tight. It provides a stricter asymptotic measure of the growth rate of an algorithm compared to Big Ω notation. While Big Ω allows for the possibility that the growth rates are the same up to constant factors, Little ω ensures that the growth rate of the function is strictly greater than the comparison function. Formally, Little ω is defined as follows:

$$f(n) \in \omega(g(n)) \iff \forall\, c > 0,\, \exists\, n_0 \geq 0 \text{ such that } \forall n \geq n_0,\, 0 \leq c \cdot g(n) < f(n).$$

- $f(n)$: The function representing the actual growth rate of the algorithm.
- $g(n)$: The comparison function, representing the lower bound on $f(n)$.
- $c > 0$: A constant that scales $g(n)$ to bound $f(n)$ strictly from below.
- n_0: A constant representing the threshold beyond which the inequality holds.

Little ω notation is used to describe situations where the growth rate of an algorithm is strictly larger than the comparison function. For example, if $f(n) = n^2$, then $f(n) \in \omega(n)$, because n^2 grows strictly faster than n as n becomes large. This notation is particularly useful when emphasizing that an algorithm's growth rate is asymptotically dominant compared to another function.

Recursion

The fundemental components of recursion are

1. A base case (or base cases)
2. A method to split the problem into smaller, similar problems.

Master Theorem

The Master Theorem provides a method for solving recurrence relations of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where:

- $a \geq 1$ is the number of subproblems in the recursion,
- $b > 1$ is the factor by which the problem size is reduced in each recursive call,
- $f(n)$ is the cost of the work done outside the recursive calls, including the cost of dividing the problem and merging results.

The theorem provides asymptotic bounds on $T(n)$ based on the relationship between $f(n)$ and $n^{\log_b a}$.

The asymptotic behavior of $T(n)$ is determined by comparing $f(n)$ with $n^{\log_b a}$:

Case 1: Polynomially Smaller Work at Each Level

If $f(n) = O(n^c)$ for some constant $c < \log_b a$, then:

$$T(n) = \Theta(n^{\log_b a})$$

Explanation: The total cost is dominated by the recursive term, meaning the majority of the work is done at the top levels of recursion.

Case 2: Balanced Work at All Levels

If $f(n) = \Theta(n^{\log_b a})$, then:

$$T(n) = \Theta(n^{\log_b a} \log n)$$

Explanation: The work done at each level is roughly the same, so the logarithmic factor arises from the accumulation of work over all levels.

Case 3: Work Dominated by the Non-Recursive Term

If $f(n) = \Omega(n^c)$ for some constant $c > \log_b a$, and if the regularity condition holds:

$$af\left(\frac{n}{b}\right) \leq c_1 f(n) \quad \text{for some } c_1 < 1 \text{ and sufficiently large } n,$$

then:

$$T(n) = \Theta(f(n))$$

Explanation: The majority of the work is done at the base level of recursion rather than at the top, meaning $f(n)$ dominates the total complexity.

Example 1: Merge Sort

The recurrence relation for Merge Sort is:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Here, $a = 2, b = 2$, and $f(n) = O(n)$. We compare $f(n)$ with $n^{\log_2 2} = n$. Since $f(n) = \Theta(n^{\log_2 2})$, Case 2 applies:

$$T(n) = \Theta(n \log n)$$

Example 2: Binary Search

The recurrence relation for Binary Search is:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

Here, $a = 1, b = 2$, and $f(n) = O(1)$. We compare $f(n)$ with $n^{\log_2 1} = n^0 = 1$. Since $f(n) = O(n^c)$ with $c = 0 < \log_2 1 = 0$, Case 1 applies:

$$T(n) = \Theta(\log n)$$

Recursion Limit

Recursive solutions can reach the max recursive depth and terminate your program early. The way to avoid this is with an iterative solution. Taking the example of the Fibonacci sequence, you could implement it as in listing 5

Listing 5: Singly Linked List Node Structure

```c
#include <stdio.h>

// Function to calculate the nth Fibonacci number using recursion
int nthFibonacci(int n){
    // Base case: if n is 0 or 1, return n
    if (n <= 1){
        return n;
    }
    // Recursive case: sum of the two preceding numbers
    return nthFibonacci(n - 1) + nthFibonacci(n - 2);
```

```
}

int main(){
    int n = 5;
    int result = nthFibonacci(n);
    printf("%d\n", result);
    return 0;
}
```

However, the runtime increases exponentially. There is a much better solution, an iterative one. For listing 6 the time complexity is $O(n)$ instead of $O(2^n)$.

In fact, the runtime increases as Φ^n, where Φ is the golden ratio.

Listing 6: Singly Linked List Node Structure

```
#include <math.h>
#include <stdio.h>

// Approximate value of golden ratio
double PHI = 1.6180339;

// Fibonacci numbers upto n = 5
int f[6] = { 0, 1, 1, 2, 3, 5 };

int fib(int n)
{
    // Fibonacci numbers for n < 6
    if (n < 6)
        return f[n];

    // Else start counting from
    // 5th term
    int t = 5, fn = 5;

    while (t < n) {
        fn = round(fn * PHI);
        t++;
    }

    return fn;
}

int main()
{
    int n = 9;
```

```
    printf("%d th Fibonacci Number = %d\n", n, fib(n));
    return 0;
}
```

Space Complexity

The space complexity of a recursive function is proportional to the depth of the computation tree, since that gives us the longest possible path (read: the highest number of frames on the call stack). For instance, if each frame has a space complexity of n, and the computation tree has a depth of $\log_2(n)$, the space complexity of the algorithm is $O(n \log(n))$.

Tail Recursion

Tail recursion is a special case of recursion where the recursive call is the last operation performed in a function before returning the result. Tail recursion can be transformed into iteration, and in some languages this is done automatically under the hood.

Listing 7: Regular Recursion

```
#include <stdio.h>

int factorial(int n) {
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
    // Multiplication happens after recursive call
}

int main() {
    printf("%d\n", factorial(5)); // Output: 120
    return 0;
}
```

Listing 8: Tail Recursion

```
#include <stdio.h>

int factorial_helper(int n, int acc) {
    if (n == 0)
        return acc;
    return factorial_helper(n - 1, n * acc);
    // Recursive call is the last operation
}
```

```
int factorial(int n) {
    return factorial_helper(n, 1);
    // Start with accumulator = 1
}

int main() {
    printf("%d\n", factorial(5)); // Output: 120
    return 0;
}
```

Listing 8 is optimized by the C compiler to perform the algorithm in listing 9.

Listing 9: Iterative Solution

```
#include <stdio.h>

int factorial(int n) {
    int acc = 1;  // Initialize the accumulator
    while (n > 0) {
        acc *= n;  // Multiply the accumulator by current n
        n--;       // Decrease n
    }
    return acc;
}

int main() {
    printf("%d\n", factorial(5)); // Output: 120
    return 0;
}
```

Stack

A stack is last in, first out. The most recent entry is "pushed" to the top of the stack, and to retrieve it we "pop" it off the stack. A natural choice of data structure for a stack is a linked list.

Queue

A queue is first in, first out. Entries are "enqueued" to insert them into the queue and "dequeued" to take them off the queue. Like a stack, queues can be implemented using linked lists.

Priority Queue

Each element in a priority queue has an associated priority. In a priority queue, elements with high priority are served before elements with low priority. An example of this is airplane loading.

Linked List

Singly Linked List

A singly linked list is a linear data structure where each node contains data and a pointer to the next node. This unidirectional structure allows efficient insertion/deletion at the head, but requires O(n) time for random access.

Listing 10: Singly Linked List Node Structure

```
struct Node {
    int data;           // Data storage
    struct Node* next; // Pointer to next node
};
```

Doubly Linked List

A doubly linked list enhances the singly linked version by including both next and previous pointers in each node. This bidirectional linkage enables forward and backward traversal at the cost of increased memory overhead.

Listing 11: Doubly Linked List Node Structure

```
struct Node {
    int data;           // Data storage
    struct Node* prev; // Pointer to previous node
    struct Node* next; // Pointer to next node
};
```

Circular Linked List

A circular linked list forms a closed loop where the tail node points back to the head. This structure can be singly or doubly linked, and is particularly useful for cyclic data processing and memory-efficient buffer implementations.

Listing 12: Circular Linked List Node Structure

```
// Singly-linked circular implementation
struct Node {
    int data;
    struct Node* next; // Last node points to head
};

// Doubly-linked circular would have:
// struct Node* prev (head's prev points to tail)
```

Binary Tree

A binary tree is a tree data structure where each node has at most two children. If each node has either two (in the case of an internal node) or zero (in the case of a leaf node), it is known as a *strictly binary tree*.

There are three methods to traverse a binary tree. Note that for each method, the only difference in implementation is where the `printf` statement is.

Preorder Traversal

Visits nodes in root-left-right order. This traversal is commonly used for creating copies of trees and prefix notation in expression trees.

Listing 13: Preorder Traversal Implementation

```
void preorder(struct Node* node) {
    if (node == NULL) return;
    printf("%d ", node->data);  // Process current node
    preorder(node->left);       // Recurse on left child
    preorder(node->right);      // Recurse on right child
}
// Time Complexity: O(n)
```

Inorder Traversal

Visits nodes in left-root-right order. Produces sorted output for binary search trees. Essential for expression trees with infix notation.

Listing 14: Inorder Traversal Implementation

```
void inorder(struct Node* node) {
    if (node == NULL) return;
    inorder(node->left);        // Recurse on left child
    printf("%d ", node->data);  // Process current node
    inorder(node->right);       // Recurse on right child
}
// Time Complexity: O(n)
```

Postorder Traversal

Visits nodes in left-right-root order. Frequently used for tree deletion and postfix notation in expression evaluation.

Listing 15: Postorder Traversal Implementation

```
void postorder(struct Node* node) {
```

```
    if (node == NULL) return;
    postorder(node->left);       // Recurse on left child
    postorder(node->right);      // Recurse on right child
    printf("%d ", node->data);   // Process current node
}
// Time Complexity: O(n)
```

Binary Search Tree

A binary search tree has the key of each internal node being greater than all the keys in the respective node's left subtree and less than the ones in its right subtree. Enables efficient search ($O(\log(n))$ average case), insertion, and deletion operations.

Figure 2 shows the binary search tree for the array `[1, 12, 36, 43, 51, 52, 83, 87]`

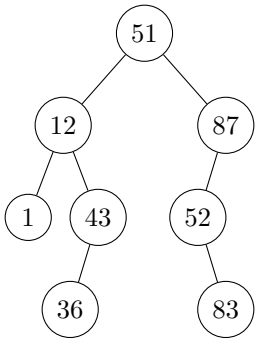

Figure 2: Binary Search Tree

Listing 16: BST Node Structure and Search

```
struct BSTNode {
    int key;
    struct BSTNode *left, *right;
};

struct BSTNode* search(struct BSTNode* root, int key) {
    if (root == NULL || root->key == key)
        return root;
    if (root->key < key)
        return search(root->right, key);
    return search(root->left, key);
}
// Time Complexity: O(h) where h = tree height
```

Key properties:

- All left descendants $\leq$ Current node $<$ All right descendants

- Inorder traversal yields sorted sequence
- Degenerate trees (completely unbalanced) degrade to $O(n)$ search

We can reconstruct a binary search tree from the preorder traversal with the algorithm in listing 17.

Listing 17: Reconstruct BST

```
Tnode *Preorder_rebuild_BST(int *a, int lidx, int ridx)
{
    if (lidx > ridx)
        return NULL;
    Tnode *root = malloc(sizeof(*root));
    if (root != NULL) {
        root->data = a[lidx];
        int partition_idx = lidx + 1;
        while (partition_idx <= ridx && a[partition_idx] < a[lidx])
            partition_idx++;
        root->left = Preorder_rebuild_BST(a, lidx+1, partition_idx-1);
        root->right = Preorder_rebuild_BST(a, partition_idx, ridx);
    }
    return root;
}
```

Each node on a binary tree has a *balance*, defined to be the difference between the height of the left substree and the height of the right subtree. If every node in a binary tree has a balance of either -1, 0, or 1, then the binary tree is height-balanced

Height-balanced binary trees are also known as **AVL** trees after Adelson-Velsky and Landis, their inventors.

When a new node is inserted into a height-balanced BST, the tree may either stay balanced or become unbalanced and need rebalancing. When the tree needs rebalancing, the damage is localized in the sense that you need only shuffle around the sibling, parent, and self nodes. The rest of the tree remains unchanged. The node that becomes unbalanced must have been either -1 or 1, which makes rebalancing the tree simple. The name of the node that could become unbalanced is the *youngest ancestor*.

In this class, the action that rebalances the tree is called a *rotation*. Rotations are either a clockwise rotation (CRs) or a counterclockwise rotation (CCR). A clockwise rotation makes the new youngest ancestor the left child, while a counterclockwise rotation makes the new youngest ancestor the right child.

Threaded Binary Tree

A *threaded binary tree* is a type of binary tree in which unused `NULL` pointers are replaced with special links called *threads*. These threads

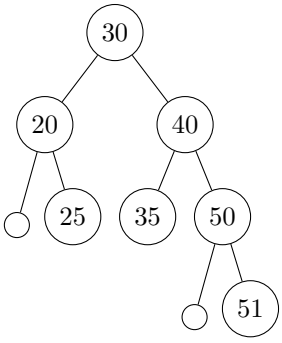

Figure 3: AVL Tree

help in performing tree traversal more efficiently by providing direct links to the in-order s uccessor (or predecessor), reducing the need for stack or recursion.

Threaded binary trees are categorized into two types:

- **Single Threaded:** Each node has a thread pointing to its *in-order successor* if the right child is `NULL`.
- **Double Threaded:** Each node has threads pointing to both its *in-order successor* and *in-order predecessor* if left or right children are `NULL`.

A typical node in a double-threaded binary tree can be represented as:

```
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
    bool lthread; // true if left is a thread
    bool rthread; // true if right is a thread
};
```

- Eliminates the need for stack or recursion during tree traversal.
- Reduces memory overhead since `NULL` pointers are replaced with useful links.
- Increases traversal efficiency, making operations like in-order traversal $\mathcal{O}(n)$ instead of $\mathcal{O}(n)$ with stack or recursion.

Since each node contains a pointer to its in-order successor, we can traverse the tree iteratively:

```
void inOrder(struct Node* root) {
    struct Node* current = root;
```

```
    // Find the leftmost node
    while (current->left != NULL && !current->lthread)
        current = current->left;

    while (current != NULL) {
        printf("%d ", current->data);

        // If right thread exists, move to successor
        if (current->rthread)
            current = current->right;
        else {
            // Else, move to the leftmost child of the right subtree
            current = current->right;
            while (current != NULL && !current->lthread)
                current = current->left;
        }
    }
}
```

Heap

A *heap* satisfies the heap property. There are two types of heaps: max-heaps and min-heaps.

- **Max-Heap**: In a max-heap, for any given node i, the value of i is greater than or equal to the values of its children. The largest element is at the root.
- **Min-Heap**: In a min-heap, for any given node i, the value of i is less than or equal to the values of its children. The smallest element is at the root.

Heaps are commonly used to implement priority queues, where the highest (or lowest) priority element is always at the root and can be accessed in constant time. The typical operations on a heap include:

- **Insertion**: Adding a new element to the heap while maintaining the heap property.
- **Deletion**: Removing the root element (the maximum in a max-heap or the minimum in a min-heap) and re-heapifying to maintain the heap property.
- **Peek**: Accessing the root element without removing it.

Heaps are usually implemented as binary heaps, which are binary trees that are complete (all levels are fully filled except possibly the last level, which is filled from left to right).

Tournaments

Consider a tournament with n players, with a normal bracket structure.

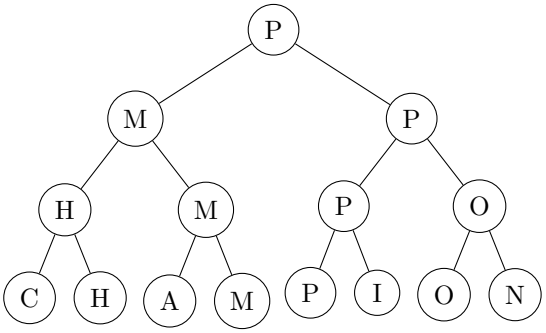

Figure 4: Tournament Tree

This tournament determines who the winner is with $n - 1$ comparisons, but it does not tell us who second is. Second could be any of the players defeated by first. To determine second, we need $\log_2(n)$ comparisons. This type of sorting is known as tournament sort, for obvious reasons, and its overall time complexity is $O(n \log(n))$.