Data Structures & Algorithms

Ryan Hou

2025-02-17

Table of contents

Preface

This is my notes on Data Structures & Algorithms in C++.

Resources

Some relevant resources:

- EECS 280 - Programming and Intro Data Structures (University of Michigan)
- EECS 281 - Data Structures and Algorithms (University of Michigan)
- EECS 376 - Foundations of Computer Science
- Data Structures & Algorithms - Google Tech Dev Guide
- EECS 281 References
- The Algorithms

Practice Resources:

- LeetCode
- NeetCode
- Blind 75
- Grind 75
  - About Grind 75
- Codeforces

Interview Resources:

- Leetcode Patterns
- Learning Resources - Reddit-wiki-programming
- AlgoMonster
- Tech Interview Handbook
  - Study Cheatsheet
- Zero To Mastery

C++ Guides:

- learncpp.com

Books:

- Algorithms - Jeff Erickson
- Cracking the Coding Interview

1 Introduction

1.1 Perspective

Note 1: Definition - Definition goes here

Definition is defined by: definition.

This notebook contains programming basics, data structures, and algorithms. The language of choice is C++, and concepts from C++ STL are also covered.

1.2 Subheading

2 Foundations

2.1 Resources

3 Pointers

4 Arrays

5 Strings

6 Streams and IO

7 Abstract Data Type

Note 2: Definition - Abstract Data Type

An **Abstract Data Type (ADT)** combines data with valid operations and their behaviors on stored data

- e.g. insert, delete, access
- ADTs define an **interface**

Meanwhile, a **data structure** provides a concrete implementation of an ADT.

8 Object-Oriented Programming

9 Dynamic Memory

10 Linked List

Linked list is a **non-contiguous** data structure with efficient **insertion** and **deletion**. Often used to implement other data structures like stacks, queues, or deques.

Main types of linked lists include:

- Singly linked list
- Doubly linked list
- Circular linked list

10.1 Summary

	Array	**Linked List**
Access	**best** time complexity $\mathbf{O(1)}$ **average** time complexity $\mathbf{O(1)}$ **worst** time complexity $\mathbf{O(1)}$	**best** time complexity $\mathbf{O(1)}$ **average** time complexity $\mathbf{O(n)}$ **worst** time complexity $\mathbf{O(n)}$
Insertion	**best** time complexity $\mathbf{O(1)}$ **average** time complexity $\mathbf{O(n)}$ **worst** time complexity $\mathbf{O(n)}$	**best** time complexity $\mathbf{O(1)}$ **average** time complexity $\mathbf{O(n)}$ **worst** time complexity $\mathbf{O(n)}$
Memory Overhead	If array size is chosen well, the array uses less memory	Additional memory is required for pointers
Memory Efficiency	May contain unused memory	Can change dynamically in size resulting in no wasted memory

10.2 C++ Example

Structure of a singly linked list node:

```
class IntList {
private:

    struct Node {
```

```
        int datum; // contains the element of the node
        Node *next; // points to the next node in the list
    }

public:
    Node *first;
    Node *last;
};
```

We can set the next pointer of the last node to `nullptr` as a sentinel value.

A more complete implementation:

```
template <typename T>
class LinkedList {
private:
    struct Node {
        T datum; // contains the element of the node
        Node *next; // points to the next node in the list
    }

    Node *first;
    Node *last;

public:
    // Constructor builds an empty list
    LinkedList() : first(nullptr) {}

    bool isEmpty() const {
        return first == nullptr;
    }

    // Return by reference the first element;
    T & front() {
        assert(!empty());
        return first->datum;
    }

    // Push a new node to the front
    void push_front(T datum) {
        Node *p = new Node;
        p->datum = datum;
```

```
        p->next = first;
        first = p;
    }

    void push_back(T datum) {
        Node *p = new Node;
        p->datum = datum;
        p->next = nullptr;
        if (empty()) {
            first = last = p;
        } else {
            last->next = p;
            last = p;
        }
    }

    // Pop the front node
    void pop_front() {
        assert(!empty());
        Node *victim = first;
        first = first->next;
        delete victim;
    }

    // Printing the linked list to os
    void print(ostream &os) const {
        for (Node *np = first; np; np = np->next) {
            os << np->datum << " ";
        }
    }

    void pop_all() {
        while (!empty()) {
            pop_front();
        }
    }

    void push_all(const LinkedList &other) {
        for (Node *np = other.first; np; np = np->next) {
            push_back(np->datum);
        }
    }
```

```
    // - The Big Three for LinkedList -

    // Destructor
    ~LinkedList() {
        pop_all();
    }

    // Copy constructor
    LinkedList(const LinkedList &other) : first(nullptr), last(nullptr) {
        push_all(other);
    }

    // Assignment Operator
    LinkedList & operator=(const LinkedList &rhs) {
        if (this == &rhs) { return *this; }
        pop_all();
        push_all(rhs);
        return *this;
    }

};
```

11 Iterators

11.1 References

- Geeks for Geeks Iterators in STL

11.2 Iterators in C++ STL

Iterator Declaration

```
<type>::iterator it;
```

Can then be initialize by assigning some valid iterator. If we already have an iterator to be assigned at the time of delcaration, then we can skip the type declaration using the auto keyword.

`auto it = iter`

Example:

```
vector<int> arr = {1, 2, 3, 4, 5};

vector<int>::iterator first = arr.begin();

vector<int>::iterator last = arr.end();

while(first != last) {
    cout << *first << " ";
    first++;
}
```

Iterator Function

Return Value

begin()

Returns an iterator to the beginning of container.

end()

Returns an iterator to the theoretical element just after the last element of the container.

cbegin()

Returns a constant iterator to the beginning of container. A constant iterator cannot modify the value of the element it is pointing to.

cend()

Returns a constant iterator to the theoretical element just after the last element of the container.

rbegin()

Returns a reverse iterator to the beginning of container.

rend()

Returns a reverse iterator to the theoretical element just after the last element of the container.

crbegin()

Returns a constant reverse iterator to the beginning of container.

crend()

Returns a constant reverse iterator to the theoretical element just after the last element of the container.

11.2.1 Iterator Operations

```
*it;            // Access
*it = new_val;  // Update
it++;           // post-increment
++it;           // pre-increment
it--;
--it;
it + int_val;   // can add or subtract by int val or another iterator
// Comparing two iterators also works (e.g. !=, <=, etc)
```

12 Recursion

12.1 Recurrence Relations

Recurrence relation describes the way a problem depends on a subproblem

12.1.1 Solving Recurrences: Linear

$$T(n) = \begin{cases} c_0, & \text{if } n = 0 \\ T(n-1) + c_1, & \text{if } n > 0 \end{cases}$$

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

Complexity: $\Theta(n)$

12.1.2 Solving Recurrences: Logarithmic

$$T(n) = \begin{cases} c_0, & \text{if } n = 0 \\ T(\frac{n}{2}) + c_1, & \text{if } n > 0 \end{cases}$$

Recurrence: $T(n) = T(n/2) + c$

Complexity: $\Theta(logn)$

12.1.3 Master Theorem

A.k.a Master Method

Let $T(n)$ be a monotonically increasing function that satisfies:

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

$T(1) = c_0$ or $T(0) = c_0$

where $a \geq 1$, $b > 1$. If $f(n) \in \Theta(n^c)$, then:

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^c \\ \Theta(n^c \log n) & \text{if } a = b^c \\ \Theta(n^c) & \text{if } a < b^c \end{cases}$$

When NOT to use Master Theorem:

- $T(n)$ is not monotonic, such as $T(n) = sin(n)$
- $f(n)$ is not a polynomial, e.g. $f(n) = 2^n$
- b cannot be expressed as a constant, e.g. $T(n) = T(\sqrt(sin(n)))$

12.1.4 Fourth Condition

A fourth condition that allows polylogarithmic functions:

13 Function Objects, Functors

14 Error Handling & Exceptions

15 Stacks, Queues, Deque, Priority Queue

15.1 Stack

15.1.1 Stack - Interface

Stack is a data structure that supports insertion/removal in Last In, First Out (LIFO) order

Stack ADT Interface:

Method	Description
`push(object)`	Add object to top of the stack
`pop()`	Remove top element
`object &top()`	Return a reference to top element
`size()`	Number of elements in stack
`empty()`	Checks if stack has no elements

15.1.2 Stack Implementation

A stack can be implemented with an array/vector or linked list

15.1.3 Stack in STL

```cpp
#include <stack>
std::stack<>
```

The underlying containers are `std::deque<>` (by default), and `std::list<>, std::vector<>` (optional).

15.2 Queue

15.2.1 Queue - Interface

Queue is a data structure that supports insertion/removal in First In, First Out (FIFO) order

15.2.2 Queue Implementation

15.2.3 Queue in STL

```
#include <queue>
std::queue<>
```

The underlying containers are `std::deque<>` (by default), and `std::list<>` (optional).

15.3 Deque

15.3.1 Deque - Interface

Deque is an abbreviation of Double-Ended Queue

```
#include <deque>
std::deque<>
```

Main methods:

- `push_front()`
- `pop_front()`
- `front()`
- `push_back()`
- `pop_back()`
- `back()`
- `size()`
- `empty()`
- Random Access: `[]` or `.at()`

STL incudes constant time `operator[]()`

15.3.2 Deque - Implementation

Circular Buffer

Doubly-linked list

15.4 Priority Queue

Each datum in the priority queue is paired with a priority value (usually numbers, should be comparable). Supports **insertion**, **inspection** of data, and **removal** of datum with highest priority.

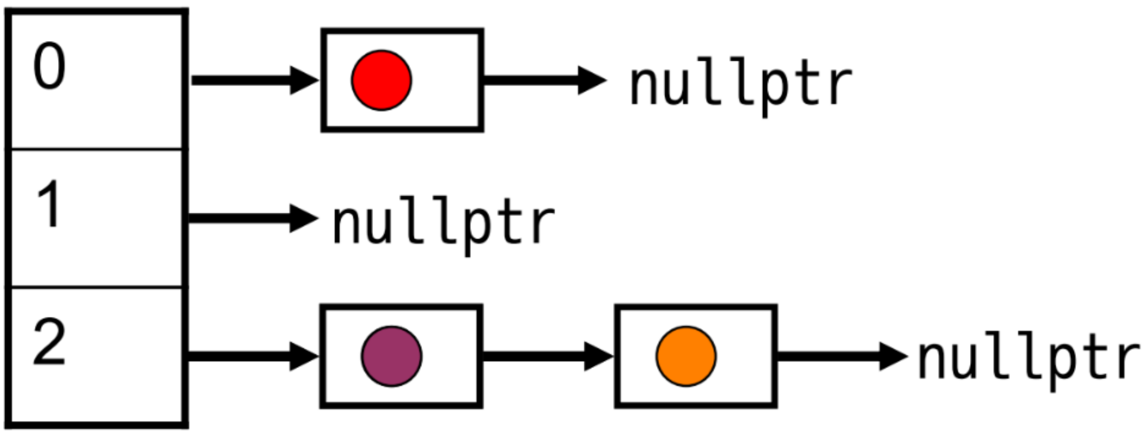

Figure 15.1: Example Priority Queue. Source: EECS 281. Lower numbers here indicate higher priority. Top element would be red node.

15.4.1 ADT - Interface

Method	Description
`push(object)`	Add object to the priority queue
`pop()`	Remove highest priority element
`const object &top()`	Return a reference to highest priority element
`size()`	Number of elements in priority queue
`empty()`	Checks if priority queue has no elements

15.4.2 Priority Queue Implementations

Priority queues can be implemented with many data structures. Heap is a common implementation.

	Insert	Remove
Unordered sequence container	Constant	Linear
Sorted sequence container	Linear	Constant
Heap	Logarithmic	Logarithmic
Array of linked lists (for priorities of small integers)	Constant	Constant

15.4.3 C++ Priority Queue

`std::priority_queue<>`

By default, uses `std::less<>` to determine priority. A default priority queue is a "max-PQ", where the largest element has highest priority. To implement a "min-PQ", use `std::greater<>`. Custom comparator (function object) needed if the elements cannot be compared with std less/greater.

Max PQ (`std::less<>`):

`std::priority_queue<T> myPQ;`

PQ with custom comparator type, `COMP`:

`std::priority_queue<T, vector<T>, COMP> myPQ;`

Manual priority queue implementation with standard library functions:

```
#include <algorithm>
std::make_heap();
std::push_heap();
std::pop_heap();
```

16 Generating Permutations

We can generate permutations by "juggling with stacks and queues"

Essentially, given N elements, we want to generate all N element permutations.

Main ingredients:

- One recursive function
- One stack
- One queue

Technique: move elements between the two containers

```
// Helper function for printing
template <typename T>
ostream &operator<<(ostream &out, const vector<T> &v) {
    // display contents of a vector on a single line
    for (auto &el : v) {
        out << el << ' ';
    }
    return out;
}

// Implementation
template <typename T>
void genPerms(vector<T> &perm, deque<T> &unused) {
    if (unused.empty()) {
        // Base case: we have reached a permutation when unused is empty
        //    i.e. a full permutation has been formed
        cout << perm << '\n';
        return;
    }

    for (size_t k = 0; k != unused.size(); ++k) {
        perm.push_back(unused.front());   // Pick the first element from unused
        unused.pop_front();               // Remove this element from unused
        genPerms(perm, unused);           // Recursively generate permutation
```

```cpp
        unused.push_back(perm.back());    // Restore this element to unused
        perm.pop_back();                  // Remove it from the permutation
    }
}

// Example Usage
int main() {
    size_t n = 16;

    vector<size_t> perm;
    deque<size_t> unused(n);
    iota(unused.begin(), unused.end(), 1); // Fills unused with consecutive numbers starting
    genPerms(perm, unused);

    return 0;
}
```

Explanation of `genPerms()`:

- The function iterates over `unused` and chooses each element one by one as it fills up a permutation
- The chosen element is moved from `unused` to `perm` (backtracking)
- The function is recursively called to generate the remaining permutation (as each call picks another element from `unused`)
- After the recursion returns, the removed element is restored to `unused`
- Time complexity: $O(n!)$ since it generates all permutations

Another Implementation of `genPerms`

```cpp
template <typename T>
void genPerms(vector<T> &path, size_t permLength) {
    if (permLength == path.size()) {
        // Do something with the path
        return;
    }
    if (!promising(path, permLength))
        return;
    for (size_t i = permLength; i < path.size(); ++i) {
        swap(path[permLength], path[i]);
        genPerms(path, permLength + 1);
        swap(path[permLength], path[i]);
    }
}
```

16.0.1 STL next_permutation()

The STL has function `std::next_permutation()`

Example Usage

```
#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    std::string s = "aba";

    do
    {
        std::cout << s << '\n';
    }
    while (std::next_permutation(s.begin(), s.end()));

    std::cout << s << '\n';
}
```

17 Complexity Analysis

17.1 References

- AlgoMonster - Runtime Summary

17.2 Runtime Overview

17.3 Common Time Complexities

Notation	Name
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Loglinear, Linearithmic
$O(n^2)$	Quadratic
$O(n^3)$, O(n), ...	Polynomial
O(c)	Exponential
O(n!)	Factorial
O(2^(2))	Doubly Exponential

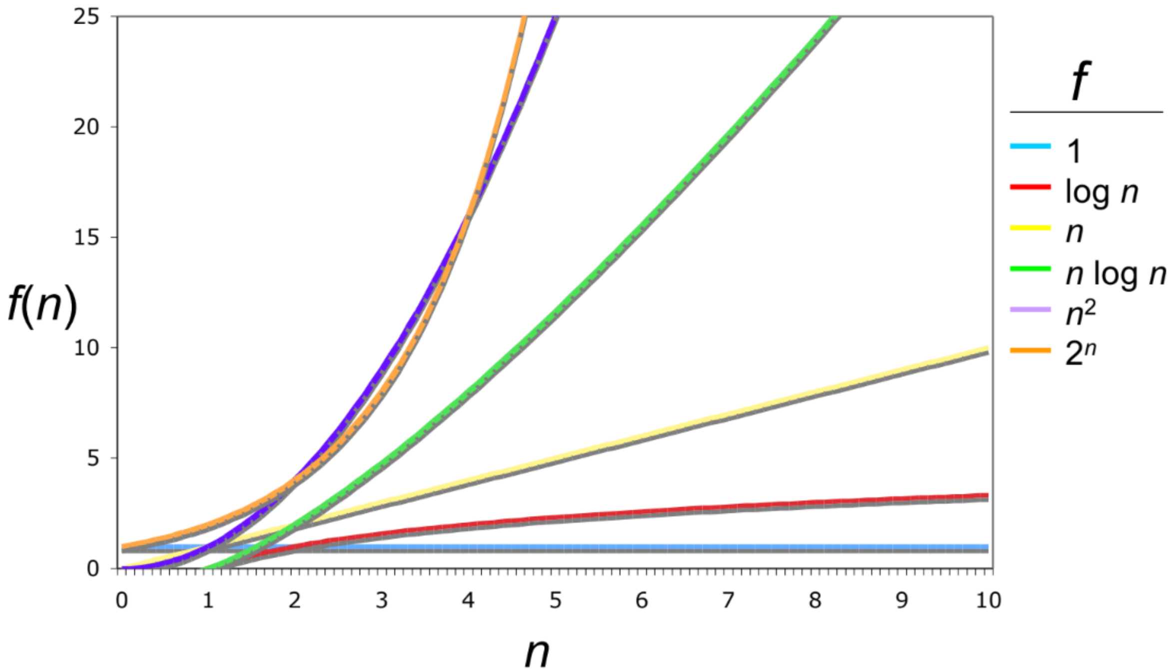

Figure 17.1: Graphing Complexities. Source: EECS 281.

17.3.1 O(1) - Constant

Constant time complexity. Could be

- Hashmap lookup
- Array access and update
- Pushing and popping elements from a stack
- Finding and applying math formula

17.3.2 O(log(N)) - Logarithmic

$log(N)$ grows very slowly

In coding interviews, log(N) typically means:

- Binary search or variant
- Balanced binary search tree lookup
- Processing the digits of a number

Unless specified, typically log(N) refers to $log_2(N)$

Example C++:

```
int N = 100000000;
while (N > 0) {
  // some constant operation
  N /= 2;
}
```

Many mainstream relational databases use binary trees for indexing by default, thus lookup by primary key in a relational database is log(N).

17.3.3 O(N) - Linear

Linear time typically means looping through a linear data structure a constant number of times. Most commonly, this means:

- Going through array/linked list
- Two pointers
- Some types of greedy
- Tree/graph traversal
- Stack/Queue

Example C++:

```
for (int i = 1; i <= N; i++) {
  // constant time code
}

for (int i = 1; i < 5 * N + 17; i++) {
  // constant time code
}

for (int i = 1; i < N + 538238; i++) {
  // constant time code
}
```

17.3.4 O(K log(N))

- Heap push/pop K times. When you encounter problems that seek the "top K elements", you can often solve them by pushing and popping to a heap K times, resulting in an O(K log(N)) runtime. e.g., K closest points, merge K sorted lists.
- Binary search K times.

Since K is constant this kind of isn't its own time complexity and can be grouped with O(log(N))

17.3.5 O(N log(N)) - Log-Linear

- Sorting. The default sorting algorithm's expected runtime in all mainstream languages is N log(N). For example, java uses a variant of merge sort for object sorting and a variant of Quick Sort for primitive type sorting.
- Divide and conquer with a linear time merge operation. Divide is normally log(N), and if merge is O(N) then the overall runtime is O(N log(N)). An example problem is smaller numbers to the right.

17.3.6 O(N^2) - Quadratic

- Nested loops, e.g., visiting each matrix entry
- Many brute force solutions

```
for (int i = 1; i <= N; i++) {
  for (int j = 1; j <= N; j++) {
    // constant time code
  }
}
```

17.3.7 O(2^N) - Exponential

Grows very rapidly. Often requires memoization to avoid repeated computations and reduce complexity.

- Combinatorial problems, backtracking, e.g. subsets
- Often involves recursion and is harder to analyze time complexity at first sight

E.g.: A recursive Fibonacci algorithm is $O(2^N)$

```
int Fib(int n) {
  if (n == 0 || n == 1) {
    return 1;
  }
  return Fib(n - 1) + Fib(n - 2);
}
```

17.3.8 O(N!) - Factorial

Grows very very rapidly. Only solvable by computers for small N. Often requires memoization to avoid repeated computations and reduce complexity.

- Combinatorial problems, backtracking, e.g. permutations
- Often involves recursion and is harder to analyze time complexity at first sight

17.4 Big-O, Big-Theta, and Big-Omega

	Big-O (O)	**Big-Theta (Θ)**	**Big-Omega (Ω)**
Defines	Asymptotic upper bound	Asymptotic tight bound	Asymptotic lower bound
Definition	$f(n) = O(g(n))$ if and only if there exists an integer n_0 and a real number c such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$	$f(n) = \Theta(g(n))$ if and only if there exists an integer n_0 and real constants c_1 and c_2 such that for all $n \geq n_0$: $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	$f(n) = \Omega(g(n))$ if and only if there exists an integer n_0 and a real number c such that for all $n \geq n_0$, $f(n) \geq c \cdot g(n)$
Mathematical Definition	$\exists n_0 \in \mathbb{Z}, \exists c \in \mathbb{R}: \forall n \geq n_0, f(n) \leq c \cdot g(n)$	$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$	$\exists n_0 \in \mathbb{Z}, \exists c \in \mathbb{R}: \forall n \geq n_0, f(n) \geq c \cdot g(n)$
$f_1(n) = 2n + 1$	$O(n)$ or $O(n^2)$ or $O(n^3)$...	$\Theta(n)$	$\Omega(n)$ or $\Omega(1)$
$f_2(n) = n^2 + n + 5$	$O(n^2)$ or $O(n^3)$...	$\Theta(n^2)$	$\Omega(n^2)$ or $\Omega(n)$ or $\Omega(1)$

17.5 Amortized Time Complexity

18 STL

19 Trees

A **graph** consists of **nodes/vertices** connected together by **edges**. Each node can contain some data.

A **tree** is

1. A connected graph (nodes + edges) without cycles
2. A graph where any 2 nodes are connected by a unique shortest path

The two definitions above are equivalent.

In a **directed tree**, we can identify **child** and **parent** relationships between nodes.

In a **binary tree**, a node has at most two children.

Terminology:

- **Root**: the topmost node in the tree
- **Parent**: Immediate predecessor of a node
- **Child**: Node where current node is parent
- **Ancestor**: parent of a parent (closer to root)
- **Descendent**: child of a child (further from root)
- **Internal node**: a node with children
- **Leaf node**: a node without children

```
template <class Item>
struct Node {       // a binary tree node
    Node *left;     // pointer to left child
    Node *right;    // pointer to right child
    Item item;      // data or KEY
}
```

Tree Properties

Height:

height(empty) = 0

height(node) = max(heght(left_child), height(right_child)) + 1

Size:

size(empty) = 0

size(node) = size(left_child) + size(right_child) + 1

Depth:

depth(empty) = 0

depth(node) = depth(parent) + 1

19.0.1 Complete (Binary) Trees

> **Note 3: Definition - Complete Binary Tree**
>
> **Complete Binary Tree:** every level, except possibly the last, is completely filled, and all nodes are as far left as possible

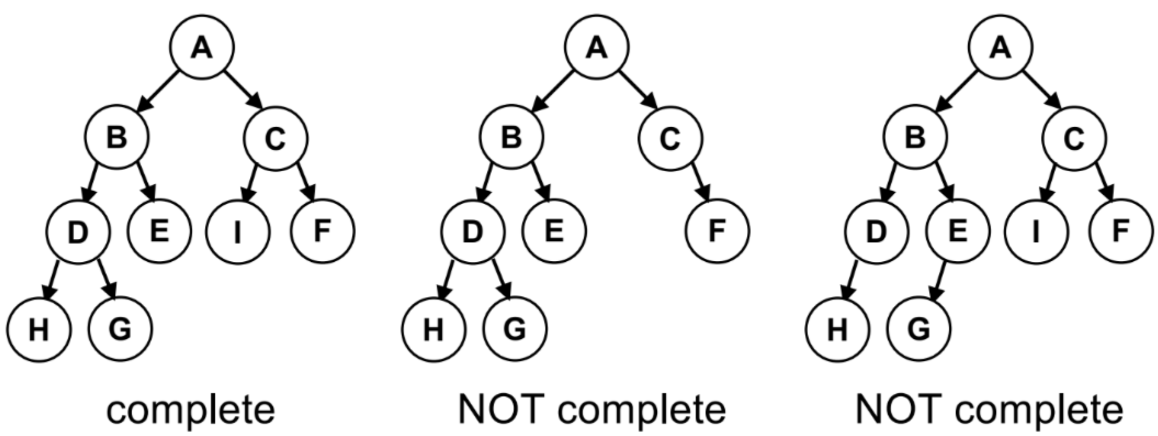

Figure 19.1: Source: EECS 281.

A complete binary tree can be stored efficiently in a growable array (vector) by indexing nodes according to level-ordering

20 Heaps

20.1 References

- AlgoMonster - Heap Fundamentals
- Geeks for Geeks - STL Heap

20.2 Heap? Priority Queue?

Priority Queue is an **Abstract Data Type**, and Heap is the concrete data structure we use to implement a priority queue. Source

Definition: A tree is (max) heap-ordered if each node's priority is not greater than the priority of the node's parent

Definition: A heap is a set of nodes with keys arranged in a complete heap-ordered tree, represented as an array

Property: No node in a heap has a key larger than the root's key

A heap has two crucial properties (representation invariants):

1. Completeness
2. Heap-ordering

These two properties can be leveraged to create an efficient priority queue and an efficient sorting algorithm using a heap!

Loose definition: data structure that gives easy access to the most extreme element, e.g., maximum or minimum

"Max Heap": heap with largest element being the "most extreme"

"Min Heap": heap with smallest element being the "most extreme"

Heaps use complete (binary) trees* as the underlying structure, but are often implemented using arrays

Note: Not to be confused with binary *search* trees.

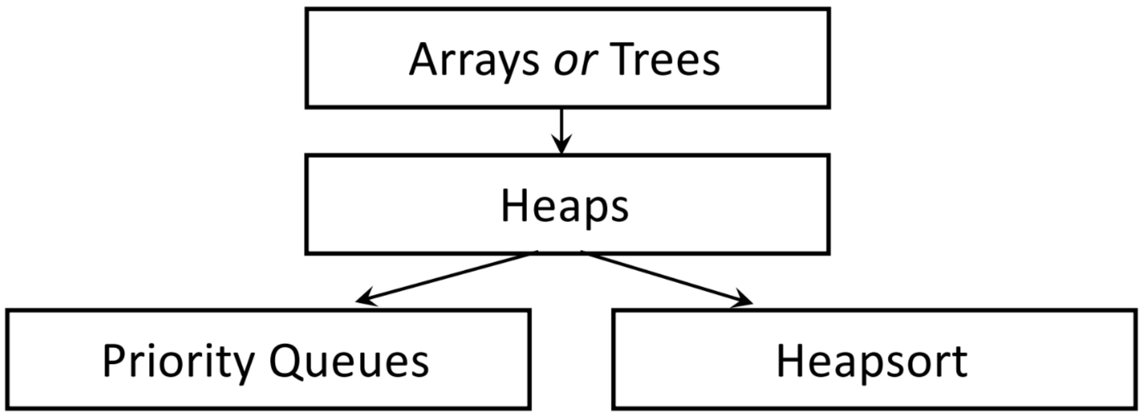

Figure 20.1: Source: EECS 281

20.3 Heap in C++

`std::make_heap()` is used to convert the given range in a container to a heap. Max heap by default. Use a custom comparator to change it to a min heap.

```
// Initializing a vector
std::vector<int> v1 = { 20, 30, 40, 25, 15 };

// Converting vector into a heap
std::make_heap(v1.begin(), v1.end());

std::cout << v1.front() << '\n'; // Displays max element of heap
```

`std::push_heap(begin_iterator, end_iterator)` sorts the heap after insertion. Can you `push_back()` of vector class to insert.

```
vector<int> vc{ 20, 30, 40, 10 };
make_heap(vc.begin(), vc.end());

vc.push_back(50);
push_heap(vc.begin(), vc.end()); // now the heap is sorted
```

Time Complexity: $O(logN)$

Note: The push_heap() function should only be used after the insertion of a single element at the back. Undefined for random insertion or for building a heap.

`pop_heap()` is used to move largest element of the heap to the end of the heap, so then a `pop_back()` can be used to remove the element

```
pop_heap(vc.begin(), vc.end()); // moves largest element to the end
vc.pop_back(); // removes element at the end
```

Time Complexity: $O(logN)$

`sort_heap()` is used to sort the heap in ascending order using heapsort.

```
sort_heap(v1.begin(), v1.end());
```

Time Complexity: $O(NlogN)$

`is_heap()` can be used to check whether a given range of the container is a heap. Default checks for max heap.

is_heap_until()

21 Searching

22 Sorting

23 Hash Maps

24 Graphs

25 Brute-Force & Greedy Algorithms

26 Divide and Conquer, Dynamic Programming

27 Backtracing, Branch and Bound Algorithms

28 Summary

In summary...

29 Tips on Solving DS&A Questions

29.1 Problem Solving Flowchart

29.2 DS&A Roadmap

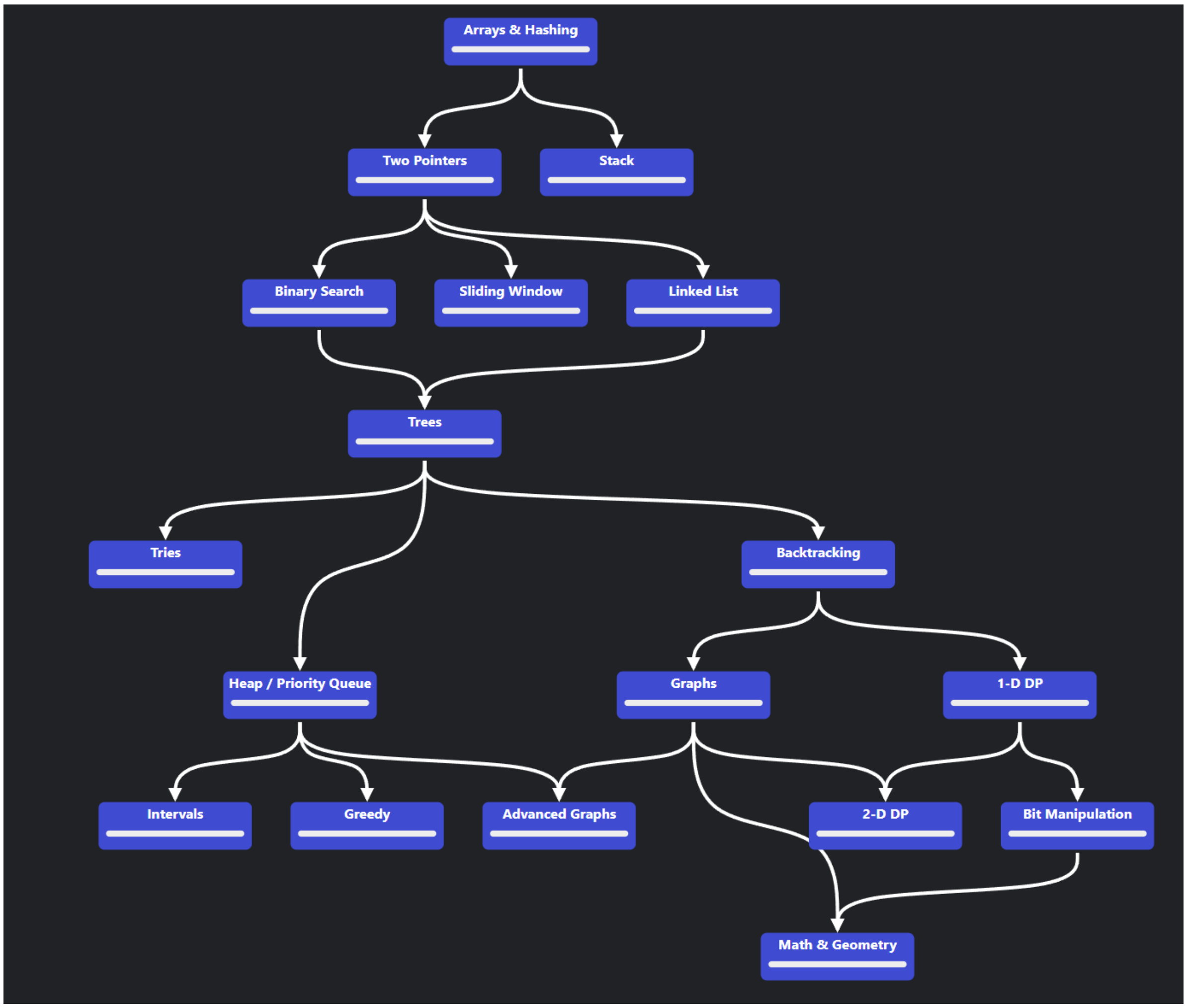

Figure 29.2: A Roadmap for studying. Source

```
If input array is sorted then
- Binary search
- Two pointers

If asked for all permutations/subsets then
- Backtracking

If given a tree then
- DFS
- BFS

If given a graph then
- DFS
- BFS

If given a linked list then
- Two pointers

If recursion is banned then
- Stack

If must solve in-place then
- Swap corresponding values
- Store one or more different values in the same pointer

If asked for maximum/minimum subarray/subset/options then
- Dynamic programming

If asked for top/least K items then
- Heap
- QuickSelect

If asked for common strings then
- Map
- Trie

Else
- Map/Set for O(1) time & O(n) space
- Sort input for O(nlogn) time and O(1) space
```

Figure 29.1: Tips on problem approach.Image Source

29.3 Problem Flowchart

29.4 ROI

29.5 "Academic" Algorithms

According to AlgoMonster, some **algorithms** that are very rarely/almost never asked in interviews:

- Minimal spanning tree: Kruskal's algorithm and Prim's algorithm
- Minimum cut: Ford-Fulkerson algorithm
- Shortest path in weight graphs: Bellman-Ford-Moore algorithm
- String search: Boyer-Moore algorithm

29.6 Keyword to Algo

AlgoMonster provides a convenient "Keyword to Algorithm" summary:

"Top k"

- Heap
  - E.g. K closest points

"How many ways.."

- DFS
  - E.g. Decode ways
- DP
  - E.g. Robot paths

"Substring"

- Sliding window
  - E.g. Longest substring without repeating characters

"Palindrome"

- two pointers: Valid Palindrome
- DFS: Palindrome Partitioning
- DP: Palindrome Partitioning II

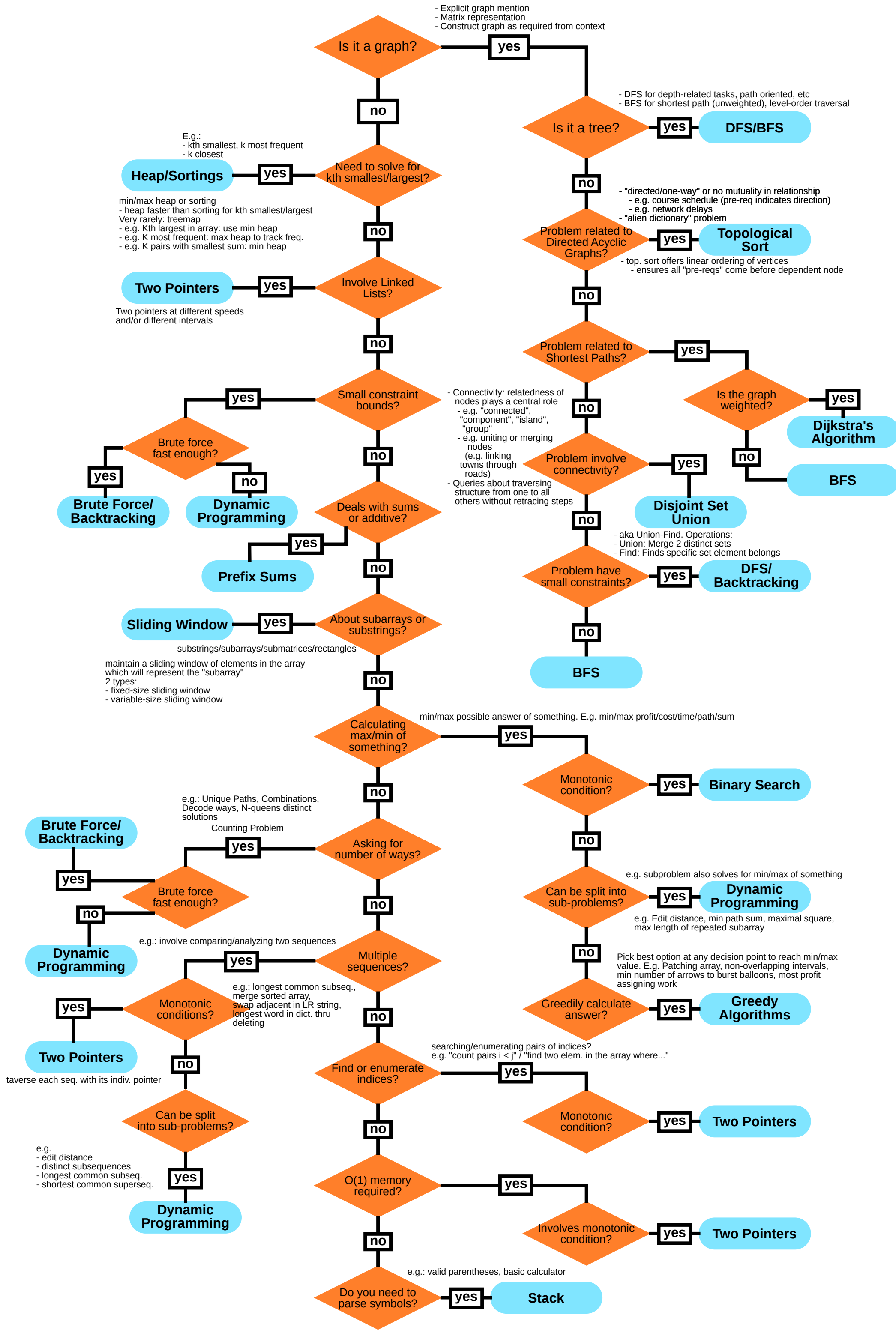

Figure 29.3: A problem solving flowchart based on AlgoMonster's flowchart. Open the SVG in a new tab and zoom in/out for better viewing.

Topic	Difficulty to Learn
Two Pointers	Easy
Sliding Window	Easy
Breadth-First Search	Easy
Depth-First Search	Medium
Backtracking	High
Heap	Medium
Binary Search	Easy
Dynamic Programming	High
Divide and Conquer	Medium
Trie	Medium
Union Find	Medium
Greedy	High

Figure 29.4: Studying to Maximizing ROI according to AlgoMonster.

"Tree"

- shortest, level-order
  - BFS: Binary Tree Level-Order Traversal
- else: DFS: Max Depth

"Parentheses"

- Stack: Valid Parentheses

"Subarray"

- Sliding window: Maximum subarray sum
- Prefix sum: Subarray sum
- Hashmap: Continuous subarray sum

Max subarray

- Greedy: Kadane's Algorithm

"X Sum"

- Two pointer: Two sum

"Max/longest sequence"

- Dynamic programming, DFS: Longest increasing subsequence
- mono deque: Sliding window maximum

"Minimum/Shortest"

- Dynamic programming, DFS: Minimal path sum
- BFS: Shortest path

"Partition/split … array/string"

- DFS: Decode ways

"Subsequence"

- Dynamic programming, DFS: Longest increasing subsequence
- Sliding window: Longest increasing subsequence

"Matrix"

- BFS, DFS: Flood fill, Islands
- Dynamic programming: Maximal square

“Jump”

- Greedy/DP: Jump game

“Game”

- Dynamic programming: Divisor game, Stone game

“Connected component”, “Cut/remove” “Regions/groups/connections”

- Union Find: Number of connected components, Redundant connections

Transitive relationship

- If the items are related to one another and the relationship is transitive, then chances are we can build a graph and use BFS or Union Find.
  - string converting to another, BFS: Word Ladder
  - string converting to another, BFS, Union Find: Sentence Similarity
  - numbers having divisional relationship, BFS, Union Find: Evaluate Division

“Interval”

- Greedy: sort by start/end time and then go through sorted intervals Interval Pattern

30 Problems & Explanations