Data structures and Algorithms

Complexity (Big O)

Complexity of an algorithm measures time and memory. How long does one algorithm takes to finish? How much memory does it use?

We measure these values in terms of the input size (Usually for the input size we use the letter n)

We can analyze algorithms in the Best case, Average case and Worst case.

Best case measures the quickest way for an algorithm to finish.

Average case measures what time does the algorithm need to finish on average.

Worst case measures the slowest time for an algorithm to finish.

General idea

1. Big O: $\mathcal{O}(n^2)$ - the algorithm takes **at most** n^2 steps to finish in the **WORST CASE**.
2. Little o: $o(n^2)$ - the algorithm takes **less than** n^2 steps to finish in the **WORST CASE**.
3. Big Omega: $\Omega(n^2)$ - the algorithm takes **at least** n^2 steps to finish in the **BEST CASE**.
4. Little Omega: $\omega(n^2)$ - the algorithm takes **more than** n^2 steps to finish in the **BEST CASE**.
5. Big Theta: $\Theta(n^2)$ - the combination of $\mathcal{O}(n^2)$ and $\Omega(n^2)$. Meaning the algorithm takes at least n^2 steps to finish and takes at most n^2 steps to finish, i.e. it always takes n^2 steps.

Comparison of orders of common functions

$$\mathcal{O}(1) < \mathcal{O}(loglog(n)) < \mathcal{O}(log(n)) < \mathcal{O}(log^2(n)) < \mathcal{O}(\sqrt{n}) < \mathcal{O}(n) < \mathcal{O}(nlog(n)) < \mathcal{O}(n^2) < \\ < \mathcal{O}(n^3) < \mathcal{O}(2^n) < \mathcal{O}(3^n) < \mathcal{O}(n!) < \mathcal{O}(n^n) < \mathcal{O}(3^{n^2}) < \mathcal{O}(2^{n^3})$$

Big O cheat sheet

https://www.bigocheatsheet.com/

Formal definitions

$$\mathcal{O}(f(n)) = \{g(n) : \exists c > 0, \exists n_0 > 0 \mid 0 \leq g(n) \leq cf(n)\ \forall n \geq n_0\}$$

$$\Theta(f(n)) = \{g(n) : \exists c_1 > 0, \exists c_2 > 0, \exists n_0 > 0 \mid 0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n)\ \forall n \geq n_0\}$$

$$\Omega(f(n)) = \{g(n) : \exists c > 0, \exists n_0 > 0 \mid 0 \leq cf(n) \leq g(n)\ \forall n \geq n_0\}$$

$$o(f(n)) = \{g(n) : \forall c > 0\ \exists n_0 > 0 \mid 0 \leq g(n) < cf(n)\ \forall n \geq n_0\}$$

$$\omega(f(n)) = \{g(n) : \forall c > 0\ \exists n_0 > 0 \mid 0 \leq cf(n) < g(n)\ \forall n \geq n_0\}$$

Examples

$\mathcal{O}(34n^4 + 5) = n^4$

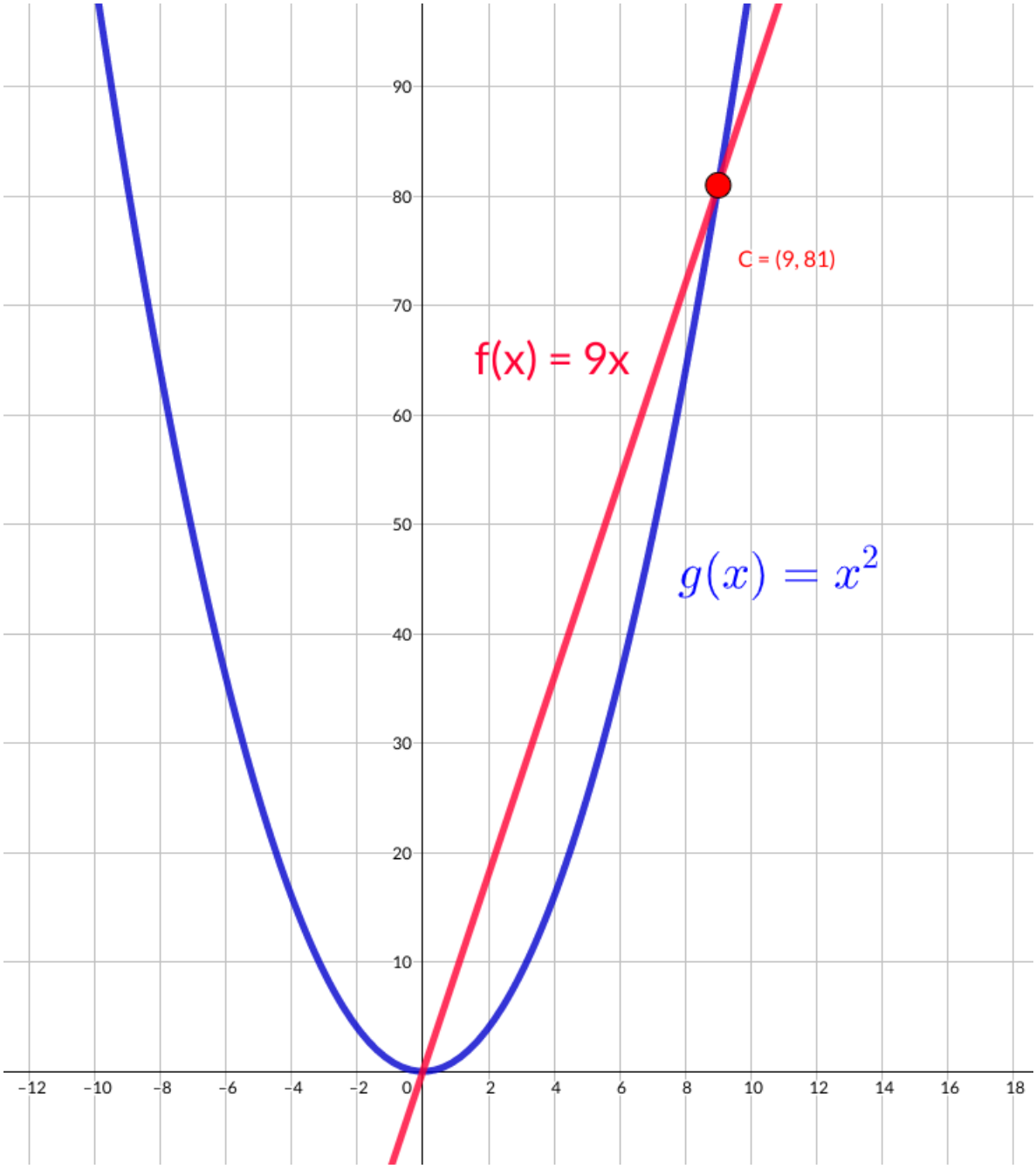
$\mathcal{O}(2^n + n^{1024} + log(n)) = 2^n$

$\mathcal{O}(5N + \frac{1}{2}M) = N + M$

$\mathcal{O}(n^2)$ means that if our algorithm takes input of 10 elements it will need 100 steps to finish in the worst case. If the input is of 100 elements it will need 10000 steps to finish in the worst case.

Constants and Big O

Usually we discard constants when using Big O, but in some cases the constant can be meaningful for small input sizes. In the following example $\mathcal{O}(n^2)$ is faster than $\mathcal{O}(n)$ because our complexities are n^2 and $9n$ so for $n < 9$ the n^2 algorithm is faster than the $9n$ one.



Sorting

Sorting means ordering a set of elements in a sequence.

We can sort a set of elements whose elements are in partial order. Partial order is a relation with the following properties:

1. Antisymmetry - For every 2 elements in the set (A, B), if $A \leq B$ and $B \leq A$ then $A = B$.
2. Transitivity - For every 3 elements in the set (A, B, C), if $A \leq B$ and $B \leq C$ then $A \leq C$.
3. Reflexivity - For every element in the set A, $A \leq A$.

Sorting properties

Stable sort

A sorting algorithm is stable if two equal objects appear in the same order in the ordered output as they appeared in the unsorted input.

> Example input: $1, 2, 3_a, 8, 5, 3_b$. Here 3_a and 3_b are simply a 3 but we have marked them to follow what happens with them after the sorting.
>
> Example output: $1, 2, 3_a, 3_b, 5, 8$
>
> Unstable sort output: $1, 2, 3_b, 3_a, 5, 8$

In place sort

Uses only a small constant amount of extra memory. In place sort means the elements are swapped around. Out of place means we use another extra array to swap items around.

Number of comparisons

How many times do we need to compare 2 elements. For most sorts this represents the time complexity.

Adaptive sort

Determines whether the sorting algorithm runs faster for inputs that are partially or fully sorted. If an algorithm is unadaptive it runs for the same time on sorted and unsorted input. If an algorithm is adaptive it runs faster on sorted input than on unsorted input.

Online sort

Determines if an algorithm needs all the items from the input to start sorting. If an algorithm is online it can start sorting input that is given in parts. If an algorithm is offline it can start sorting only after it has the whole input.

External sort

Allows sorting data which cannot fit into memory. Such algorithms are designed to sort massive amounts of data (Gigabytes, Terabytes, etc.)

Parallel sort

An algorithm which can be ran on multiple threads at the same time, speeding the running time.

Locality

An algorithm which heavily uses the processor cache to speed up its execution. The data processed needs to be sequentially located.

Helper code

```
// swap the values of two variables
void swap(int & a, int & b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

Bubble sort

Bubble sort	n = input size
Time complexity	$\mathcal{O}(n^2)$
Space complexity	$\mathcal{O}(1)$
Adaptive	Yes
Stable	Yes
Number of comparisons	$\mathcal{O}(n^2)$
Number of swaps	$\mathcal{O}(n^2)$
Online	No
In place	Yes

Honorable mention - Cocktail shaker sort, similar to bubble and selection sort. It's like the Online version of bubble sort.

Clean code

```
void bubbleSort(int * array, int length) {
    for (int bubbleStartIndex = 0; bubbleStartIndex < length;
bubbleStartIndex++) {
        for (int bubbleMovedIndex = 0; bubbleMovedIndex < length - 1;
bubbleMovedIndex++) {
            if (array[bubbleMovedIndex] > array[bubbleMovedIndex+1]) {
                swap(array[bubbleMovedIndex], array[bubbleMovedIndex+1]);
            }
        }
    }
}
```

Short code

```
void bubbleSort(int * array, int length) {
    for (int i = 0; i < length; i++) {
        for (int k = 0; k < length - 1; k++) {
            if (array[k] > array[k+1]) {
                swap(array[k], array[k+1]);
            }
        }
    }
}
```

Optimization 1 - make it faster

```
void bubbleSort(int * array, int length) {
    for (int i = 0; i < length; i++) {
        for (int k = 0; k < length - 1 - i; k++) { // length - 1 - i = 2x less
iterations
            if (array[k] > array[k+1]) {
                swap(array[k], array[k+1]);
            }
        }
    }
}
```

Optimization 2 - make it adaptive

```
void bubbleSort(int * array, int length) {
    for (int i = 0; i < length; i++) {
        bool swappedAtLeastOnce = false; // add a flag to check if a swap has
occurred

        for (int k = 0; k < length - 1 - i; k++) {
            if (array[k] > array[k+1]) {
                swap(array[k], array[k+1]);
                swappedAtLeastOnce = true;
            }
        }

        if (!swappedAtLeastOnce) { // if there were no swaps, it's ordered
            break;
        }
    }
}
```

Selection sort

Selection sort	n = input size
Time complexity	$\mathcal{O}(n^2)$
Space complexity	$\mathcal{O}(1)$
Adaptive	No
Stable	No
Number of comparisons	$\mathcal{O}(n^2)$
Number of swaps	$\mathcal{O}(n)$
Online	No
In place	Yes

Key points

1. It's better than bubble sort in cases we need to sort items which are very large in size because it has $\mathcal{O}(n)$ swaps.

Clean code

```
void selectionSort(int * array, int length) {
    for (int currentIndex = 0; currentIndex < length; currentIndex++) {
        int smallestNumberIndex = currentIndex;

        for (int potentialSmallerNumberIndex = currentIndex + 1;
                potentialSmallerNumberIndex < length;
                potentialSmallerNumberIndex++) {
            if (array[potentialSmallerNumberIndex] < array[smallestNumberIndex])
{
                smallestNumberIndex = potentialSmallerNumberIndex;
            }
        }

        swap(array[currentIndex], array[smallestNumberIndex]);
    }
}
```

Short code

```
void selectionSort(int * array, int length) {
    for (int i = 0; i < length; i++) {
        int index = i;

        for (int k = i + 1; k < length; k++) {
            if (array[k] < array[index]) {
                index = k;
            }
        }

        swap(array[i], array[index]);
    }
}
```

Insertion sort

Insertion sort	n = input size
Time complexity	$\mathcal{O}(n^2)$
Space complexity	$\mathcal{O}(1)$
Adaptive	Yes
Stable	Yes
Number of comparisons	$\mathcal{O}(n^2)$
Number of swaps	$\mathcal{O}(n^2)$
Online	Yes
In place	Yes

Key points

1. The best algorithm for sorting small arrays.
2. Often combined with other algorithms for optimal runtime.

Clean code

```
void insertionSort(int * array, int length) {
    for (int nextItemToSortIndex = 1; nextItemToSortIndex < length;
nextItemToSortIndex++) {
        for (int potentiallyBiggerItemIndex = nextItemToSortIndex;
                potentiallyBiggerItemIndex > 0 &&
                array[potentiallyBiggerItemIndex] <
array[potentiallyBiggerItemIndex - 1];
                potentiallyBiggerItemIndex--) {
            swap(array[potentiallyBiggerItemIndex],
array[potentiallyBiggerItemIndex-1]);
        }
    }
}
```

Short code

```
void insertionSort(int * array, int length) {
    for (int i = 1; i < length; i++) {
        for (int k = i; k > 0 && array[k] < array[k - 1]; k--) {
            swap(array[k], array[k-1]);
        }
    }
}
```

Merge sort

Merge sort	n = input size
Time complexity	$\mathcal{O}(n * log(n))$
Space complexity	$\mathcal{O}(n)$
Adaptive	No
Stable	Yes
Number of comparisons	$\mathcal{O}(n * log(n))$
External	Yes
Parallel	Yes
Online	No
In place	No

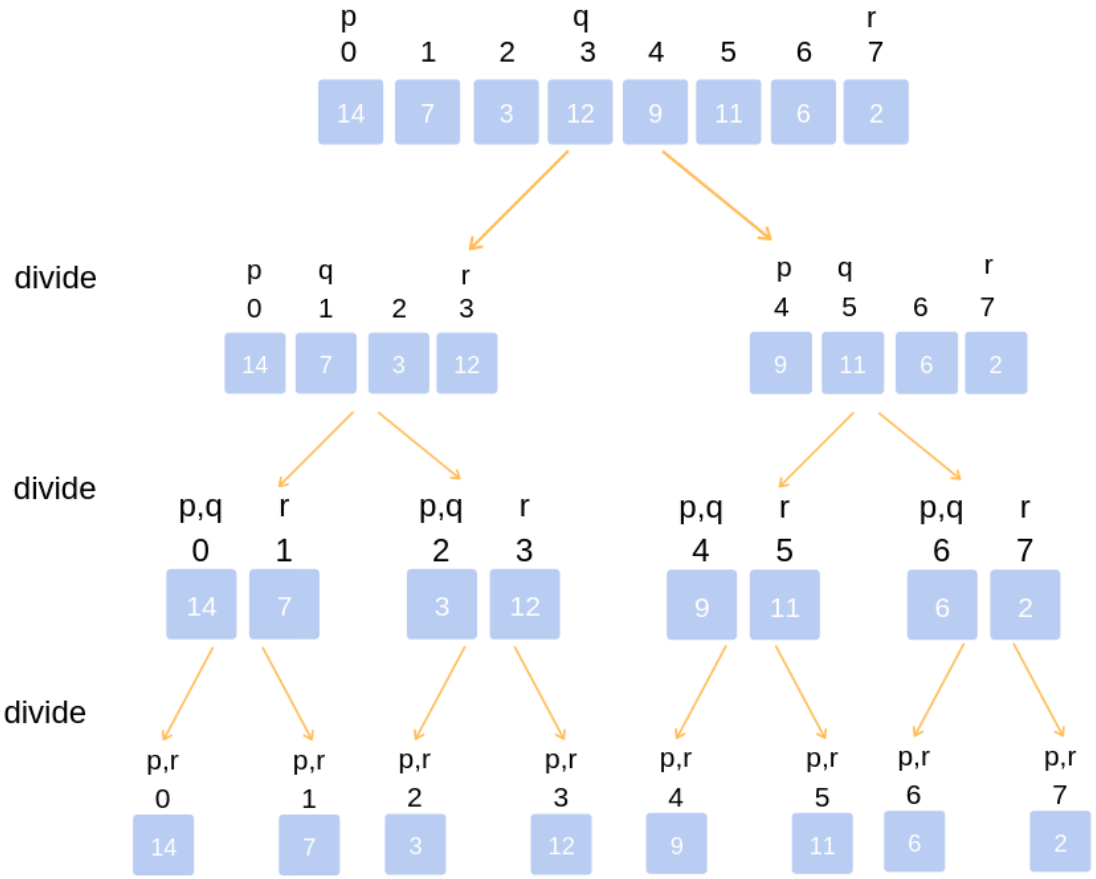
Key points

1. Every array of 0 or 1 elements is sorted.
2. Merging requires additional memory.
3. Splitting the array in 2 every time gives us $log(n)$ levels.

4. On each level we have $\mathcal{O}(n)$ for the merging operation.

Honorable mention - Ford–Johnson aka. Merge-insertion sort. Cool idea, but not practical enough.

Important algorithm: Timsort - Absolute beast with $\mathcal{O}(nlogn)$ worst case and $\mathcal{O}(n)$ best case. Implementation can be found here: https://github.com/tvanslyke/timsort-cpp



Clean code

```
void merge(int * originalArray, int * mergeArray, int start, int mid, int end) {
    int leftIndex = start;
    int rightIndex = mid + 1;
    int sortedIndex = start;

    while (leftIndex <= mid && rightIndex <= end) {
        if (originalArray[leftIndex] <= originalArray[rightIndex]) {
            mergeArray[sortedIndex] = originalArray[leftIndex];
            leftIndex++;
        } else {
            mergeArray[sortedIndex] = originalArray[rightIndex];
            rightIndex++;
        }

        sortedIndex++;
    }

    while (leftIndex <= mid) {
        mergeArray[sortedIndex] = originalArray[leftIndex];
        leftIndex++;
        sortedIndex++;
    }
```

```
    while (rightIndex <= end) {
        mergeArray[sortedIndex] = originalArray[rightIndex];
        rightIndex++;
        sortedIndex++;
    }

    for (int i = start; i <= end; i++) {
        originalArray[i] = mergeArray[i];
    }
}

void mergeSortRecursive(int * original, int * mergeArray, int start, int end) {
    if (start < end) {
        int mid = (start + end) / 2;

        mergeSortRecursive(original, mergeArray, start, mid);
        mergeSortRecursive(original, mergeArray, mid + 1, end);

        merge(original, mergeArray, start, mid, end);
    }
}

void mergeSort(int * array, int length) {
    int * mergeArray = new int[length];
    mergeSortRecursive(array, mergeArray, 0, length - 1);
    delete[] mergeArray;
}
```

Short code

```
void merge(int * originalArray, int * mergeArray, int start, int mid, int end) {
    int left = start;
    int right = mid + 1;

    for (int i = start; i <= end; i++) {
        if (left <= mid && (right > end || originalArray[left] <=
originalArray[right])) {
            mergeArray[i] = originalArray[left];
            left++;
        } else {
            mergeArray[i] = originalArray[right];
            right++;
        }
    }

    for (int i = start; i <= end; i++) {
        originalArray[i] = mergeArray[i];
    }
}

void mergeSortRecursive(int * originalArray, int * mergeArray, int start, int
end) {
    if (start < end) {
        int mid = (start + end) / 2;

        mergeSortRecursive(originalArray, mergeArray, start, mid);
        mergeSortRecursive(originalArray, mergeArray, mid + 1, end);
```

```
        merge(originalArray, mergeArray, start, mid, end);
    }
}

void mergeSort(int * array, int length) {
    int * mergeArray = new int[length];
    mergeSortRecursive(array, mergeArray, 0, length - 1);
    delete[] mergeArray;
}
```

Quick sort

Quick sort	n = input size
Time complexity (Worst case)	$\mathcal{O}(n^2)$
Time complexity (Average case)	$\mathcal{O}(n * log(n))$
Space complexity	$\mathcal{O}(log(n))$
Adaptive	No
Anti-Adaptive	Yes (the more randomness, the better)
Stable	No
Number of comparisons	$\mathcal{O}(n * log(n))$
Parallel	Yes
Online	No
In place	Yes
Locality	Yes

Key points

1. Quick sort adores chaos. That's why a common strategy is to shuffle the array before sorting it.

C++ STL sorting algorithm is Introsort, which is a hybrid between quicksort and heapsort.

Quick sort optimization for arrays with many equal numbers - three-way partitioning, aka Dutch national flag.

Quick sort can also be optimized with picking 2 pivots instead of 1 - multi-pivot quicksort.

Why is quick sort usually faster than merge sort and other fast algorithms?

The table bellow is an approximation of the time required to access data from the disk/ram/CPU cache. It gives us an idea why quick sort is indeed quick. Quick sort being in-place gets reduced to problems in the CPU cache very fast. Merge sort on the other hand uses additional memory, so it has 2 arrays it works with essentially. Often these 2 arrays cannot simultaneously enter the cache so they have to be read from RAM, which is very slow. Quick sort, however, uses only one array which gets pulled into the CPU cache and operations on it are very fast.

Memory hierarchy	CPU cycles	size
HDD	500, 000	1 TB
RAM	100	4 GB
L2 cache	10	512 kb
L1 cache	1	32 kb

Uses apart from sorting

1. Find k-th biggest/smallest element

Clean code using Lomuto partitioning

```
int partition(int * array, int startIndex, int endIndex) {
    int pivot = array[endIndex];
    int pivotIndex = startIndex;

    for (int i = startIndex; i <= endIndex; i++) {
        if (array[i] < pivot) {
            swap(array[i], array[pivotIndex]);
            pivotIndex++;
        }
    }

    swap(array[endIndex], array[pivotIndex]);
    return pivotIndex;
}

void _quickSort(int * array, int startIndex, int endIndex) {
    if (startIndex < endIndex) {
        int pivot = partition(array, startIndex, endIndex);
        _quickSort(array, startIndex, pivot - 1);
        _quickSort(array, pivot + 1, endIndex);
    }
}

void quickSort(int * array, int length) {
    _quickSort(array, 0, length - 1);
}
```

Short code using Lomuto partitioning

```
int partition(int * array, int start, int end) {
    int pivot = array[end];
    int pivotIndex = start;
    for (int i = start; i <= end; i++) {
        if (array[i] < pivot) {
            swap(array[i], array[pivotIndex]);
            pivotIndex++;
        }
    }
    swap(array[end], array[pivotIndex]);
    return pivotIndex;
```

```
}

void _quickSort(int * array, int start, int end) {
    if (start < end) {
        int pivot = partition(array, start, end);
        _quickSort(array, start, pivot - 1);
        _quickSort(array, pivot + 1, end);
    }
}

void quickSort(int * array, int length) {
    _quickSort(array, 0, length - 1);
}
```

Optimization 1 - pivot is random number

```
#include <random>
int generateRandomNumber(int from, int to) {
    std::random_device dev;
    std::mt19937 rng(dev());
    std::uniform_int_distribution<std::mt19937::result_type> dist6(from,to);

    return dist6(rng);
}

int partition(int * array, int start, int end) {
    int randomIndex = generateRandomNumber(start, end);
    swap(array[end], array[randomIndex]);

    int pivot = array[end];
    int pivotIndex = start;
    for (int i = start; i <= end; i++) {
        if (array[i] < pivot) {
            swap(array[i], array[pivotIndex]);
            pivotIndex++;
        }
    }
    swap(array[end], array[pivotIndex]);
    return pivotIndex;
}

void _quickSort(int * array, int start, int end) {
    if (start < end) {
        int pivot = partition(array, start, end);
        _quickSort(array, start, pivot - 1);
        _quickSort(array, pivot + 1, end);
    }
}

void quickSort(int * array, int length) {
    _quickSort(array, 0, length - 1);
}
```

Optimization 2 - shuffle before picking pivot

```
#include <random>
```

```
int generateRandomNumber(int from, int to) {
    std::random_device dev;
    std::mt19937 rng(dev());
    std::uniform_int_distribution<std::mt19937::result_type> dist6(from,to);

    return dist6(rng);
}

void shuffle(int * arr, int length) {
    for (int i = 0; i < length - 1; i++) {
        int swapCurrentWith = generateRandomNumber(i, length - 1);
        swap(arr[i], arr[swapCurrentWith]);
    }
}

int partition(int * array, int start, int end) {
    shuffle(array + start, end - start);

    int pivot = array[end];
    int pivotIndex = start;
    for (int i = start; i <= end; i++) {
        if (array[i] < pivot) {
            swap(array[i], array[pivotIndex]);
            pivotIndex++;
        }
    }
    swap(array[end], array[pivotIndex]);
    return pivotIndex;
}

void _quickSort(int * array, int start, int end) {
    if (start < end) {
        int pivot = partition(array, start, end);
        _quickSort(array, start, pivot - 1);
        _quickSort(array, pivot + 1, end);
    }
}

void quickSort(int * array, int length) {
    _quickSort(array, 0, length - 1);
}
```

Short code using Hoare partitioning

```
int partition(int * array, int start, int end)  {
    int pivot = array[(start + end) / 2];
    int left = start - 1;
    int right = end + 1;

    while (true) {
        do {
            left++;
        } while (array[left] < pivot);

        do {
            right--;
        } while (array[right] > pivot);
```

```
        if (left >= right) {
            return right;
        }

        swap(array[left], array[right]);
    }
}

void _quickSort(int * array, int start, int end) {
    if (start < end) {
        int pivot = partition(array, start, end);
        _quickSort(array, start, pivot - 1);
        _quickSort(array, pivot + 1, end);
    }
}

void quickSort(int * array, int length) {
    _quickSort(array, 0, length - 1);
}
```

Counting sort

Counting sort	n = input size, k = max number
Time complexity (Worst case)	$\mathcal{O}(n+k)$
Time complexity (Best case)	$\mathcal{O}(n)$
Space complexity	$\mathcal{O}(k)$

Key points

1. Not a comparison sort.
2. Very efficient for an array of integers with many repeating integers with small difference between the biggest and smallest integer.

Uses apart from sorting

1. Counting inversions

Counting sort for positive numbers

```
void countingSort(int * array, int length) {
    int maxNumber = 0;
    for (int i = 0; i < length; i++) {
        if (array[i] > maxNumber) {
            maxNumber = array[i];
        }
    }

    int * countingArray = new int[maxNumber + 1];
    for (int i = 0; i < maxNumber + 1; i++) {
        countingArray[i] = 0;
    }
```

```
    for (int i = 0; i < length; i++) {
        countingArray[array[i]]++;
    }

    int sortedIndex = 0;
    for (int i = 0; i < length; i++) {
        while (countingArray[sortedIndex] == 0) {
            sortedIndex++;
        }

        array[i] = sortedIndex;
        countingArray[sortedIndex]--;
    }

    delete[] countingArray;
}
```

Bucket sort

Bucket sort	n = input size, k = number of buckets
Time complexity (Worst case)	$\mathcal{O}(n^2)$
Time complexity (Average case)	$\mathcal{O}(n + \frac{n^2}{k} + k)$
Space complexity	$\mathcal{O}(nk)$

Key points

1. It's a distribution sort, not a comparison sort.
2. Very situational sort.
3. The bellow code is a sample implementation for floats between 0 and 1.

Code

```
#include <iostream>
#include <vector>
#include <algorithm>
void bucketSort(double *array, int length) {
    std::vector<double> buckets[length];

    for(int i = 0; i < length; i++) {
        buckets[int(length * array[i])].push_back(array[i]);
    }

    for(int i = 0; i < length; i++) {
        sort(buckets[i].begin(), buckets[i].end());
    }

    int index = 0;
    for(int i = 0; i < length; i++) {
        while(!buckets[i].empty()) {
            array[index] = buckets[i][0];
            index++;
            buckets[i].erase(buckets[i].begin());
        }
```

```
    }
}
```

Radix sort

Radix sort	n = input size
Time complexity (Worst case)	$\mathcal{O}(n^2)$
Time complexity (Average case)	$\mathcal{O}(n + \frac{n^2}{k} + k)$
Space complexity	$\mathcal{O}(nk)$

Key points

1. Good on multi-threaded machines.

Code

```
#include <iostream>
#include <cmath>
#include <list>

int abs(int a) {
    return a >= 0 ? a : -a;
}

void radixSort(int * array, int length) {
    int maxNumber = 0;
    for (int i = 0; i < length; i++) {
        if (abs(array[i]) > maxNumber) {
            maxNumber = abs(array[i]);
        }
    }

    int maxDigits = log10(maxNumber) + 1;

    std::list<int> pocket[10];

    for(int i = 0; i < maxDigits; i++) {
        int m = pow(10, i + 1);
        int p = pow(10, i);

        for(int j = 0; j < length; j++) {
            int temp = array[j] % m;
            int index = temp / p;
            pocket[index].push_back(array[j]);
        }

        int count = 0;
        for(int j = 0; j<10; j++) {
            while(!pocket[j].empty()) {
                array[count] = *(pocket[j].begin());
                pocket[j].erase(pocket[j].begin());
                count++;
            }
```

```
            }
        }
    }
```

Searching

Linear search

Linear search	n = input size
Time complexity	$\mathcal{O}(n)$
Space complexity	$\mathcal{O}(1)$

Algorithm idea

Iterate every element in order and check if it is the element we are looking for.

Code

```c
int linearSearch(int *array, int length, int target) {
    for (int i = 0; i < length; i++) {
        if (array[i] == target) {
            return i;
        }
    }
    return -1;
}
```

Uses

Searching in unordered data, when we have very few queries. If we have many queries it's better to sort the array and use faster searching for the queries.

Binary search

Binary search	n = input size
Time complexity	$\mathcal{O}(log(n))$
Space complexity	$\mathcal{O}(1)$

Works only on sorted array-like structures, which support $\mathcal{O}(1)$ access to elements.

The algorithm uses $\mathcal{O}(log(n))$ space if written recursively, but can be written iteratively to have $\mathcal{O}(1)$ space as well.

Algorithm idea

1. Check the middle element, if it's the element we are looking for - we are done
2. If the element is bigger than the one we are looking for - recurse into the left sub-array (the sub-array with smaller elements).
3. If the element is smaller than the one we are looking for - recurse into the right sub-array (the sub-array with bigger elements).

Code

```
int binarySearch(int *array, int length, int target) {
    int left = 0;
    int right = length - 1;

    while(left <= right) {
        int mid = (left + right) / 2;

        if (array[mid] == target) {
            return mid;
        }
        if (array[mid] > target) {
            right = mid - 1;
        }
        if (array[mid] < target) {
            left = mid + 1;
        }
    }

    return -1;
}
```

Uses

Fast searching in ordered data.

Guess and check algorithms.

Ternary search

Ternary search	n = input size
Time complexity	$\mathcal{O}(log(n))$
Space complexity	$\mathcal{O}(1)$

Algorithm idea

The same idea as binary search but we split the array in 3 parts. On each step we decide which one part to discard and recurse into the other 2.

Code

```
int ternarySearch(int *array, int length, int target) {
    int left = 0;
    int right = length - 1;

    while (left <= right) {
        int leftThird = left + (right - left) / 3;
        int rightThird = right - (right - left) / 3;

        if (array[leftThird] == target) {
            return leftThird;
        }
        if (array[rightThird] == target) {
```

```
            return rightThird;
        }

        if (target < array[leftThird]) {
            right = leftThird - 1;
        } else if (target > array[rightThird]) {
            left = rightThird + 1;
        } else  {
            left = leftThird + 1;
            right = rightThird - 1;
        }
    }

    return -1;
}
```

Uses

Finding min/max in a sorted array.

Exponential search

Exponential search	n = input size
Time complexity	$\mathcal{O}(log(n))$
Space complexity	$\mathcal{O}(1)$

Algorithm idea

This is the binary search algorithm applied for ordered data of unknown size. We don't have a mid point so we start from the beginning and double our guessed index each time. e.g for guessed index: 1, 2, 4, 8, 16, 32... If we overshoot the end of the data we go back using binary search because we now have an end index. e.g for array with 27 elements we do a jump to element 32 which doesn't exist and then use binary search on elements 16 - 32.

Code

```
int exponentialSearch(int *array, int length, int target) {
   if (length == 0) {
      return false;
   }

   int right = 1;
   while (right < length && array[right] < target) {
      right *= 2;
   }

   int binaryIndex = binarySearch(array + right/2, min(right + 1, length) -
right/2, target);
   if (binaryIndex != -1) {
      return binaryIndex + right/2;
   } else {
      return -1;
   }
}
```

Uses

Binary search on streams, which are ordered and other ordered data of unknown size.

Jump search

Jump search	n = input size
Time complexity	$\mathcal{O}(log(n))$
Space complexity	$\mathcal{O}(1)$

Algorithm idea

We decide on a jump step and check elements on each step. e.g for step 32 we will check elements 0, 32, 64, etc.

Optimal jump step is $\sqrt{n}$.

Code

```
int jumpSearch(int *array, int length, int target) {
   int left = 0;
   int step = sqrt(length);
   int right = step;

   while(right < length && array[right] <= target) {
      left += step;
      right += step;
      if(right > length - 1)
         right = length;
   }

   for(int i = left; i < right; i++) {
      if(array[i] == target)
         return i;
   }

   return -1;
}
```

Uses

Because it's worse than Binary search, it's useful when the data structure we are searching in has very slow jump back, aka it's slow to go back to index $i - 10$ from i.

Interpolation search

Interpolation search	n = input size
Time complexity for evenly distributed data	$\mathcal{O}(log(log(n)))$
Time complexity for badly distributed data	$\mathcal{O}(n)$
Space complexity	$\mathcal{O}(1)$

Algorithm idea

If we have well distributed data we can approximate where we will find the element with the following formula

$$jumpToIndex = startIndex + (\frac{endIndex - startIndex}{array[endIndex] - array[startIndex]}) \times (target - array[startIndex])$$

Example for well distributed data: 1, 2, 3, 4, 5, 6 .... 99, 100

Example for badly distributed data: 1, 1, 1, 1.... 1, 1, 100

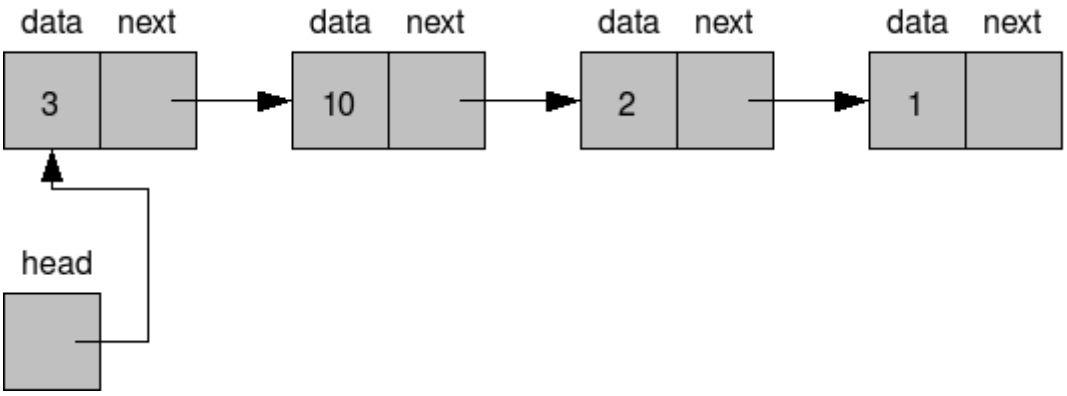
Code

```
int interpolationSearch(int *array, int length, int target) {
    int left = 0;
    int right = length - 1;

    while (array[right] != array[left] && target >= array[left] && target <=
array[right]) {
        int mid = left + ((target - array[left]) * (right - left) /
(array[right] - array[left]));

        if (array[mid] < target)
            left = mid + 1;
        else if (target < array[mid])
            right = mid - 1;
        else
            return mid;
    }

    if (target == array[left])
        return left;
    else
        return -1;
}
```

Uses

Very situational, when we have data which is evenly distributed.

Linked lists



Linked list stores its data in separate Nodes. Each node points the next node in the list. The last node points NULL.

The most primitive linked list's node contains 2 fields - Data and Next pointer

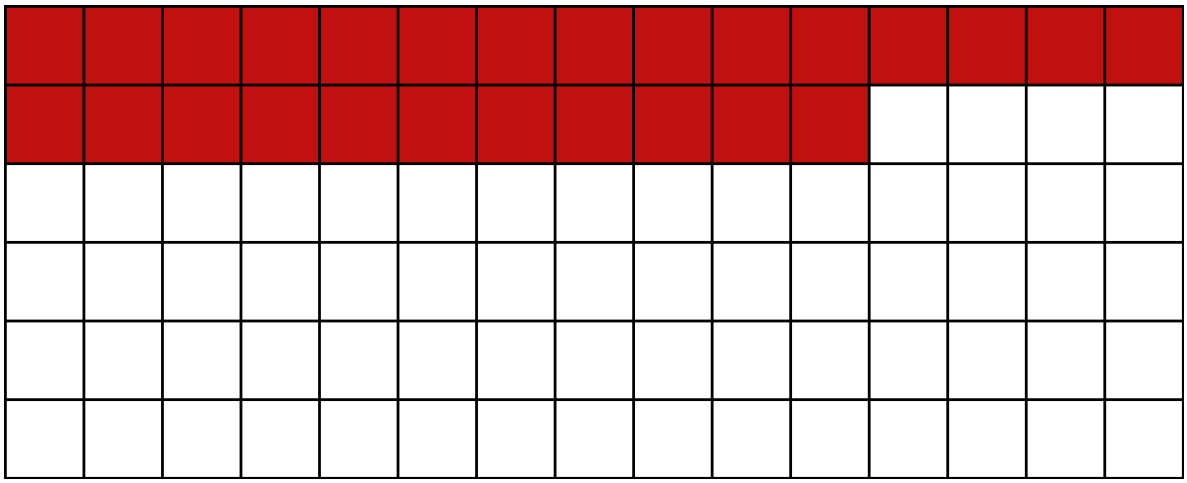
```
struct Node {
    int data;
    Node* next;
};
```

Complexity comparison

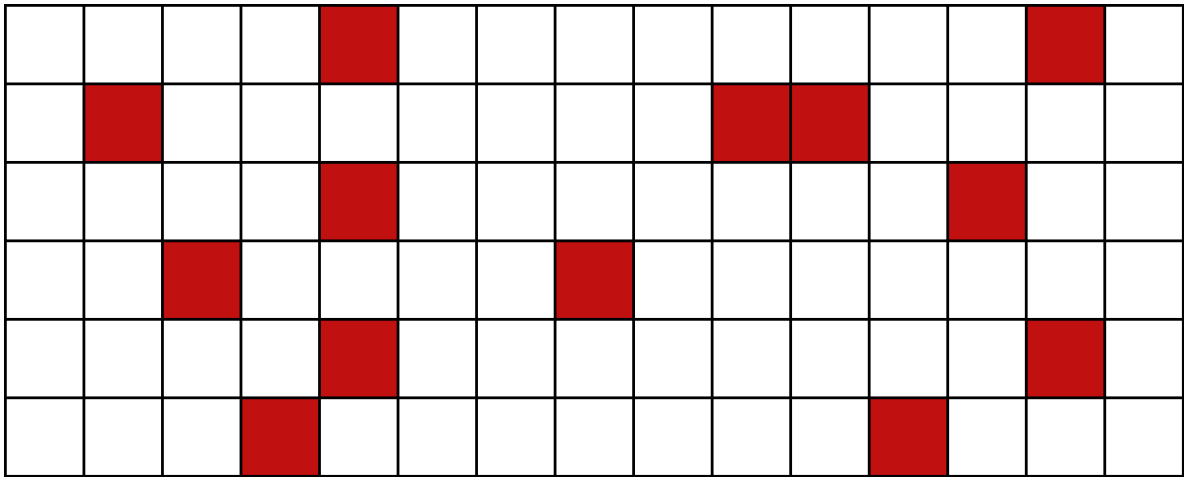
Operation / Data structure	Array	Singly linked list without tail	Singly linked list with tail	Doubly linked list without tail	Doubly linked list with tail
push_front	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
pop_front	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
get_front	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
push_back	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
pop_back	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
get_back	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
get_at	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
find_key	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
erase_key	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
is_empty	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
add_before	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
add_after	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Memory allocation

Array



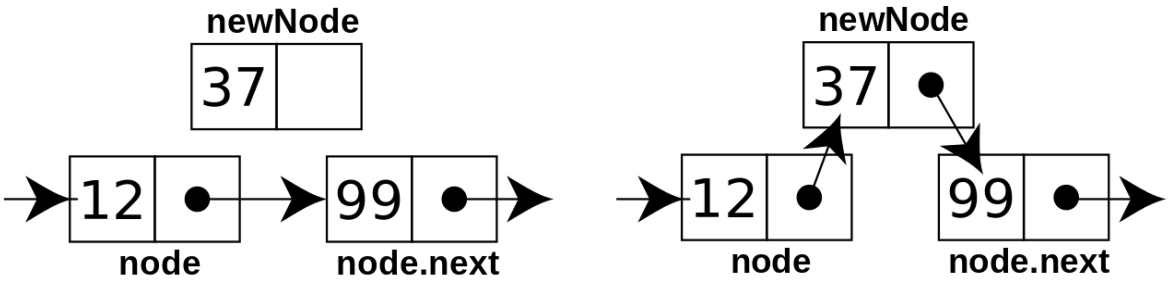
Linked list



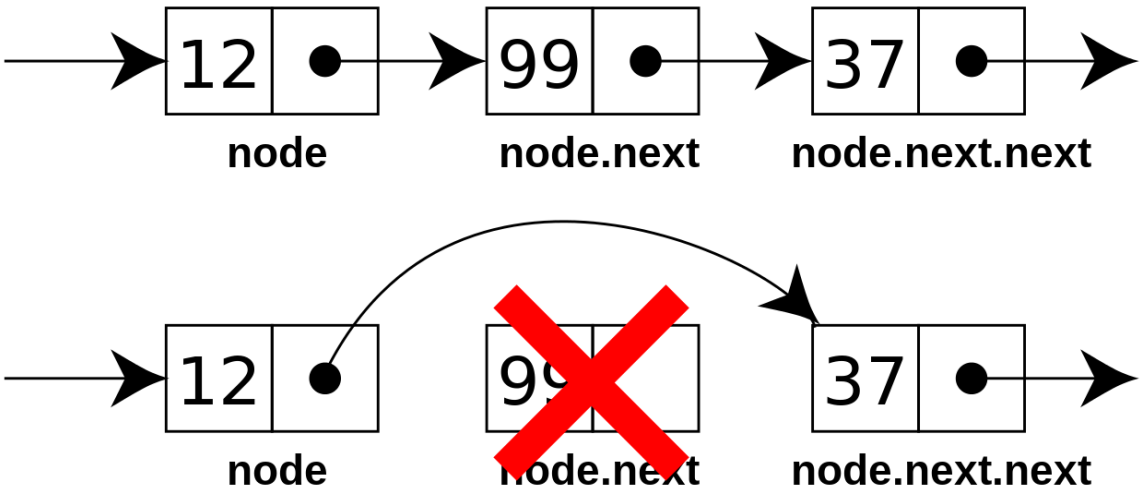
Use

Linked lists are useful if we don't know the input size (when getting data from streams).

Inserting a node



Removing a node



Singly linked list

Singly linked lists have only one pointer to the next Node.

```
struct Node {
    int data;
    Node* next;
};
```

Doubly linked list

Doubly linked lists have two pointers, one to the previous item, one to the next.

```
struct Node {
    int data;
    Node* previous;
    Node* next;
};
```

Circular linked list

Circular linked lists can be implemented as singly or doubly linked list but the start and end nodes are connected. i.e the end node's next pointer is not NULL but points the first node instead.

XOR linked list

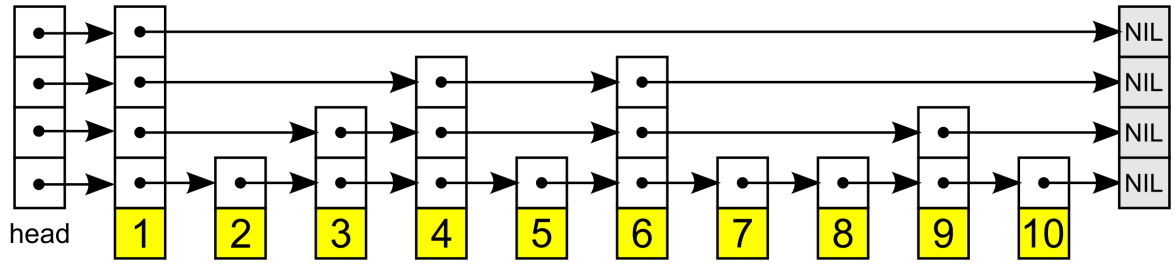
Doubly linked list using memory for singly linked list. The idea is to XOR the address of the previous element and the next element and save that XOR value in one pointer. When we need to access the next element, we XOR the pointer we have saved with the address of the previous element and the result will be the pointer to the next element.

Unrolled linked list

Linked list of arrays. Usually the size of the array is the size of the CPU cache. This way we can quickly iterate through the arrays in the cache and we also have the benefits of a linked list.

Skip list

A skip list is a linked list-like data structure which allows $\mathcal{O}(logn)$ search and $\mathcal{O}(logn)$ insertion in an ordered sequence of elements. It builds $\mathcal{O}(logn)$ lanes up with links more and more sparse. The chance to build the next lane of links up is 1/2. The higher lanes are "express lanes" for the lanes bellow. The idea is to quickly get to the approximate location we need to get to on the higher lanes and then go down on the lower lanes to the exact location we need.



Dynamic arrays

A dynamic array is an array which grows in size when it's full. Each time it increases its size 2 times. e.g the size at the start is 1, then 2, 4, 8, 16....

The good points are we have amortized $\mathcal{O}(1)$ Add and $\mathcal{O}(1)$ access like in regular arrays.

The downsides are If we have added 32 items and have reserverd 64 items, but don't add any more items we are wasting memory needlessly. Also, when adding, if the array is full we will need to resize so occasionally our Add will be with $\mathcal{O}(n)$ complexity.

Amortized complexity

Amortized complexity is the total expense per operation, evaluated over a sequence of operations.

In other words if we add 1 item it could be very expensive $\mathcal{O}(n)$ but if we add `n` items it is going be $\mathcal{O}(n)$ to add all n-items.

Explained with example:

Let's take a dynamic array (in C++ that would be std::vector).

At the start we have 1 empty slot. In constant time we add 1 item at the end.

When we want to add 1 more item, our array is full, so we expand it 2x in size to 2 items, we copy the old 1 item at the start and in constant time we add the new item at the end. The copy operation takes $\mathcal{O}(n)$ steps.

We want to add a 3rd item, but we have capacity of 2, we expand the array to 4 in $\mathcal{O}(n)$ steps and add the new item to the end in constant time.

We want to add a 4th item, and we have capacity of 4 so we can add it to the end in constant time.

In conclusion, every time we want to add an item in position 2^k we ahve to expand with complexity $\mathcal{O}(2^k)$. All other additions are with $\mathcal{O}(1)$ comlexity.

So in the end we have $1 + 2^0 + 1 + 2^1 + 1 + 1 + 2^2 + 1 + 1 + \ldots + 2^k$ where $k = log(n)$. If we sum these numbers we will get n from the 1s and 2n from the powers of 2. In conclusion adding n items costs us $\mathcal{O}(n)$ time. In other words amortized $\mathcal{O}(1)$ per item.

Stack (LIFO)

A stack is a **restricted** data structure - allows access only to the element at the top of the stack.

Add elements to the top, take from the top.

LIFO = Last In First Out

Problems that can occur with stacks

Overflow - Adding too many elements and the stack cannot contain them

Underflow - Try to access the top element when the stack is empty

Stack with	Dynamic Array	Linked list
push()	Amortized $\mathcal{O}(1)$	$\mathcal{O}(1)$
pop()	Amortized $\mathcal{O}(1)$	$\mathcal{O}(1)$
peek()/top()	$\mathcal{O}(1)$	$\mathcal{O}(1)$
clear()	$\mathcal{O}(p)$ - delete memory	$\mathcal{O}(n)$
isEmpty()	$\mathcal{O}(1)$	$\mathcal{O}(1)$
initialize	$\mathcal{O}(p)$ - allocate memory	$\mathcal{O}(1)$
merge	$\mathcal{O}(n)$	$\mathcal{O}(1)$

Application of stacks

1. Undo/Redo in programs
2. Reverse word/array
3. Programming languages are usually compiled down and work as a pushdown automaton
4. Syntax parsing - used by compilers
5. Switching between infix and prefix notation
6. Backtracking

Queue (FIFO)

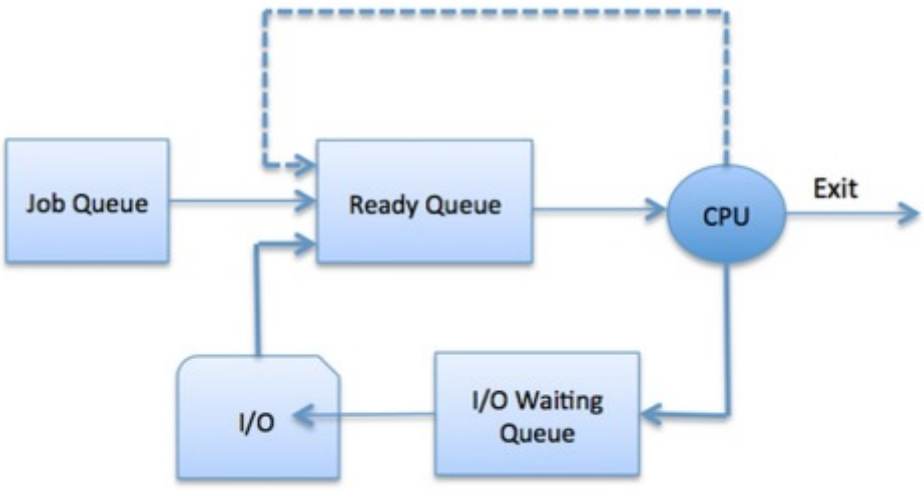
Add elements to the back, take from the front.

FIFO = First In First Out

Stack with	Dynamic Array	Linked list
push()/enqueue()	Amortized $\mathcal{O}(1)$	$\mathcal{O}(1)$
pop()/dequeue()	Amortized $\mathcal{O}(1)$	$\mathcal{O}(1)$
peek()/top()	$\mathcal{O}(1)$	$\mathcal{O}(1)$
clear()	$\mathcal{O}(p)$ - delete memory	$\mathcal{O}(n)$
isEmpty()	$\mathcal{O}(1)$	$\mathcal{O}(1)$
initialize	$\mathcal{O}(p)$ - allocate memory	$\mathcal{O}(1)$

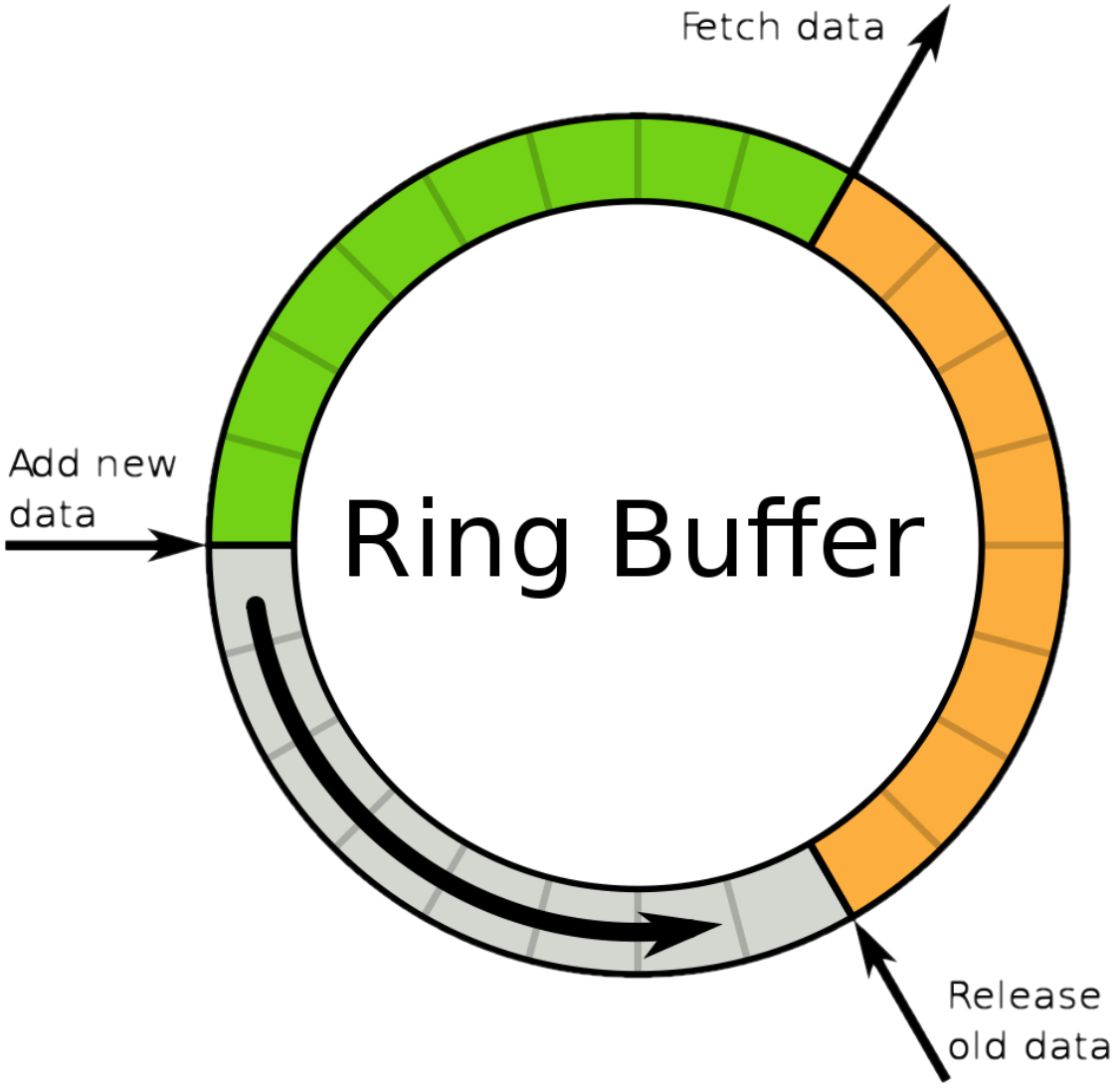
Application of queues

1. System processes
2. Interactive processes
3. Background processes
4. Batch process
5. CPU scheduling
6. Synchronize data transfer
7. Print spooler
8. First-come-first-serve (servers for example)



Ring buffer

Ring buffers are queues where the new data can overwrite the old data when needed. It's like a queue implemented with an array, providing limited space. Problem that can occur if the pointer for adding new data reaches the pointer for fetching data, in other words the front pointer and the back pointer must never overlap.



Application of ring buffers

1. Chats
2. Real-time processing

Okasaki Queue

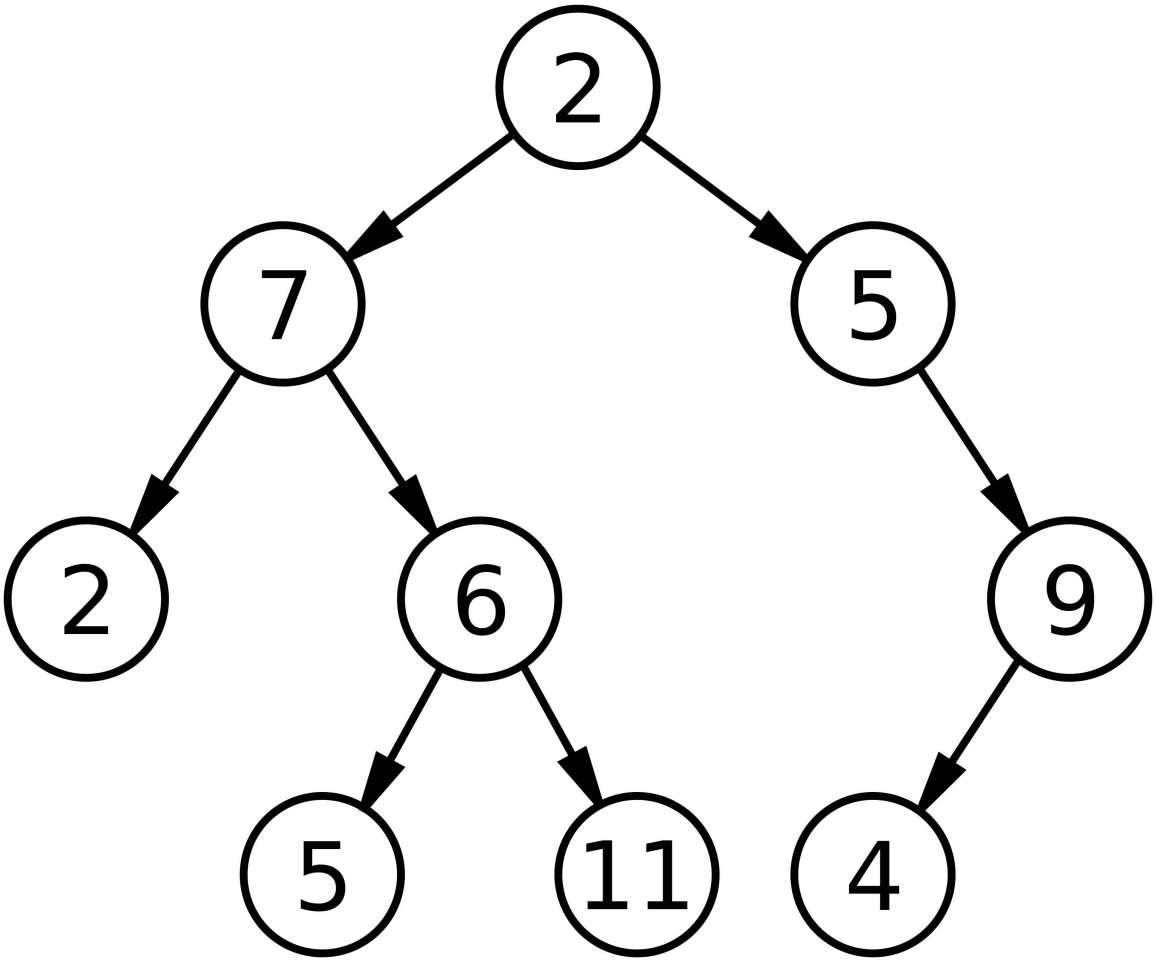
A queue implemented with 2 linked lists.

https://ucsd-progsys.github.io/liquidhaskell-tutorial/09-case-study-lazy-queues.html

Other interesting topic to research https://en.wikipedia.org/wiki/Persistent_data_structure

Trees

A tree is a hierarchical data structure. A tree has one root node (one starting element) and all other elements are children/grandchildren/etc. of this root element.



The root in the picture above is 2. Its children are 7 and 5. 5 is a child of the root 2 and also the parent of 9.

How does the tree stay connected

There are 2 ways to keep the Nodes in the tree connected.

1. Each parent has pointers to its children. (The classic widely used approach)
2. Each node has a pointer to its parent. (Not widely used in most algorithms, but still very useful in some particular problems)
3. 1 and 2 combined - Each node has pointers to its children and a pointer to its parent. This makes the traversal of the tree very easy in each direction, but also takes more memory.

Node of a binary tree

```
struct Node {
    int data;
    Node* left = nullptr;
    Node* right = nullptr;
};
```

Node of a tree with variable number of children

```
struct Node {
    int data;
    vector<Node*> children;
};
```

Traversals

Preorder (DFS)

Preorder traversal is also known as DFS (Depth First Search).

Time complexity $\mathcal{O}(n)$, where `n` is the number of nodes in the tree.

The order in which we visit nodes is

1. Current node
2. Children of the node in order from left to right

```
void DFS(Node* current) {
    if (current == nullptr) {
        return;
    }

    cout << current->data; // do something with the node

    DFS(current->left);
    DFS(current->right);
}
```

Postorder

Time complexity $\mathcal{O}(n)$, where `n` is the number of nodes in the tree.

The order in which we visit nodes is

1. Children of the node in order from left to right
2. Current node

```
void Postorder(Node* current) {
    if (current == nullptr) {
        return;
    }

    Postorder(current->left);
    Postorder(current->right);

    cout << current->data; // do something with the node
}
```

Inorder traversal

Time complexity $\mathcal{O}(n)$, where `n` is the number of nodes in the tree.

The order in which we visit nodes is

1. Left child
2. Current node
3. Right child

```
void Postorder(Node* current) {
    if (current == nullptr) {
        return;
    }

    Postorder(current->left);
    cout << current->data; // do something with the node
    Postorder(current->right);
}
```

Level order

Level order is also known as WFS (Width First Search) or BFS (Breadth First Search).

Time complexity $\mathcal{O}(n)$, where `n` is the number of nodes in the tree.

The order in which we visit nodes is

1. All the nodes of level 1 (the root)
2. All the nodes of level 2 from left to right (the children of the root)
3. All the nodes of level 3 from left to right
4. etc.

Morris order

Morris order is an inorder traversal algorithm which does not use extra space for recursion or a stack. However it changes the tree during traversal and then reverts it to its original state.

https://en.wikipedia.org/wiki/Tree_traversal#Morris_in-order_traversal_using_threading

Tree variations

Generalized tree

A tree where each node stores whatever data we need and has an arbitrary number of children.

Trie

Segment Tree

Merge Tree

K-D Tree

BSP Tree

Merkle Tree

Treap

Threaded Binary Tree

Strict Binary Tree

Full Binary Tree

Complete Binary Tree

Binary Search Trees

BST

AVL Tree

Splay Tree

Scape Goat Tree

Red-black Tree

Heap

Heap sort

Disjoint Set

Hashes

Rolling hash

Bloom Filter

Graphs

Graph variations

Undirected Graph

Directed Graph

Weighted Graph

Connected Graph

Disconnected Graph

Trees

Complete Graph

Tournament Graph

Bipartite Graph

Representing Graphs

Edge List

Adjacency Matrix

Adjacency List

Exploring Graphs

Depth First Search

Breadth First Search

Topological Sorting

Minimum Spanning Tree

Prim

Kruskal

Shortest Path

Dijkstra

Bellman-Ford

A*

NP-problems