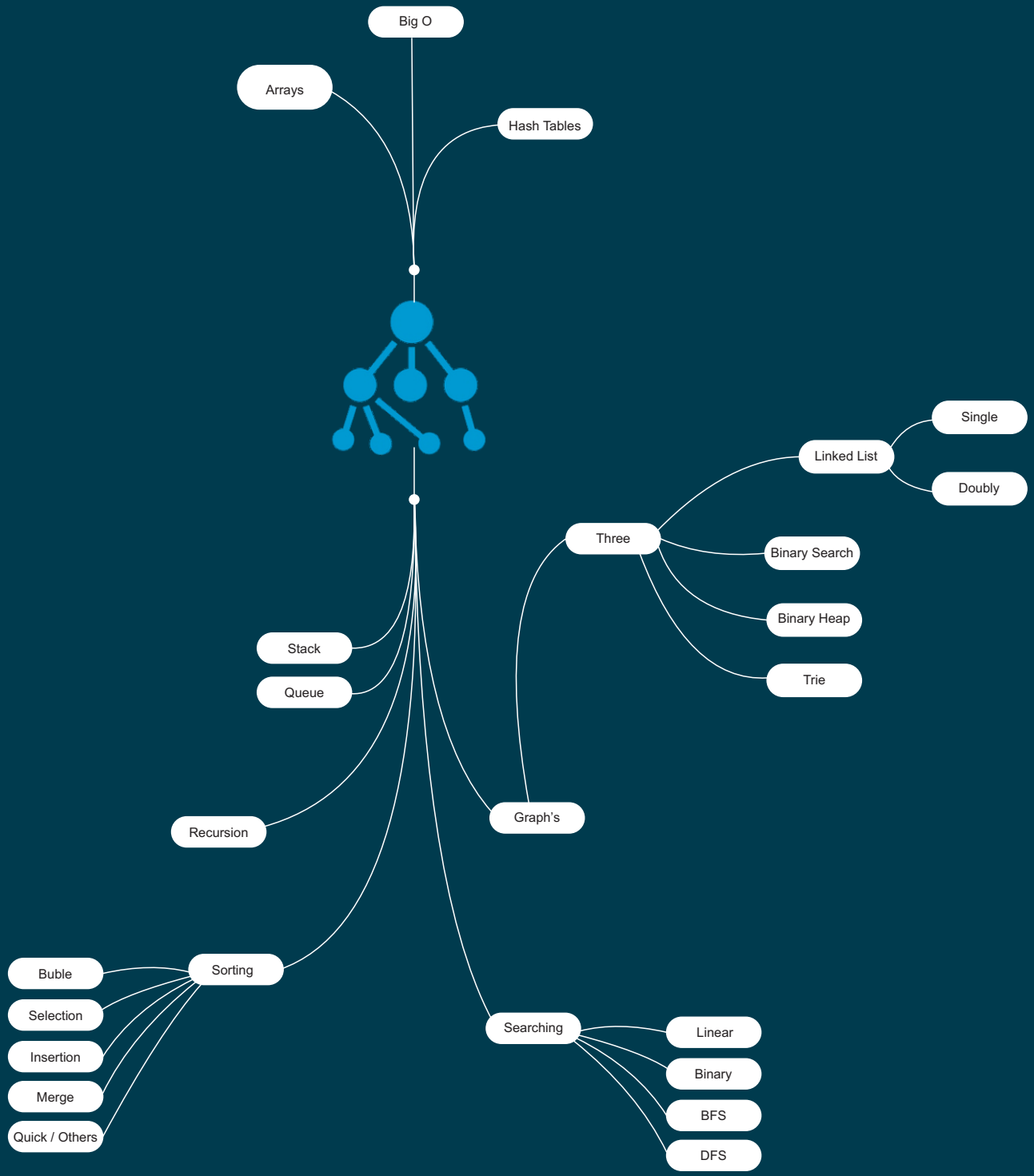
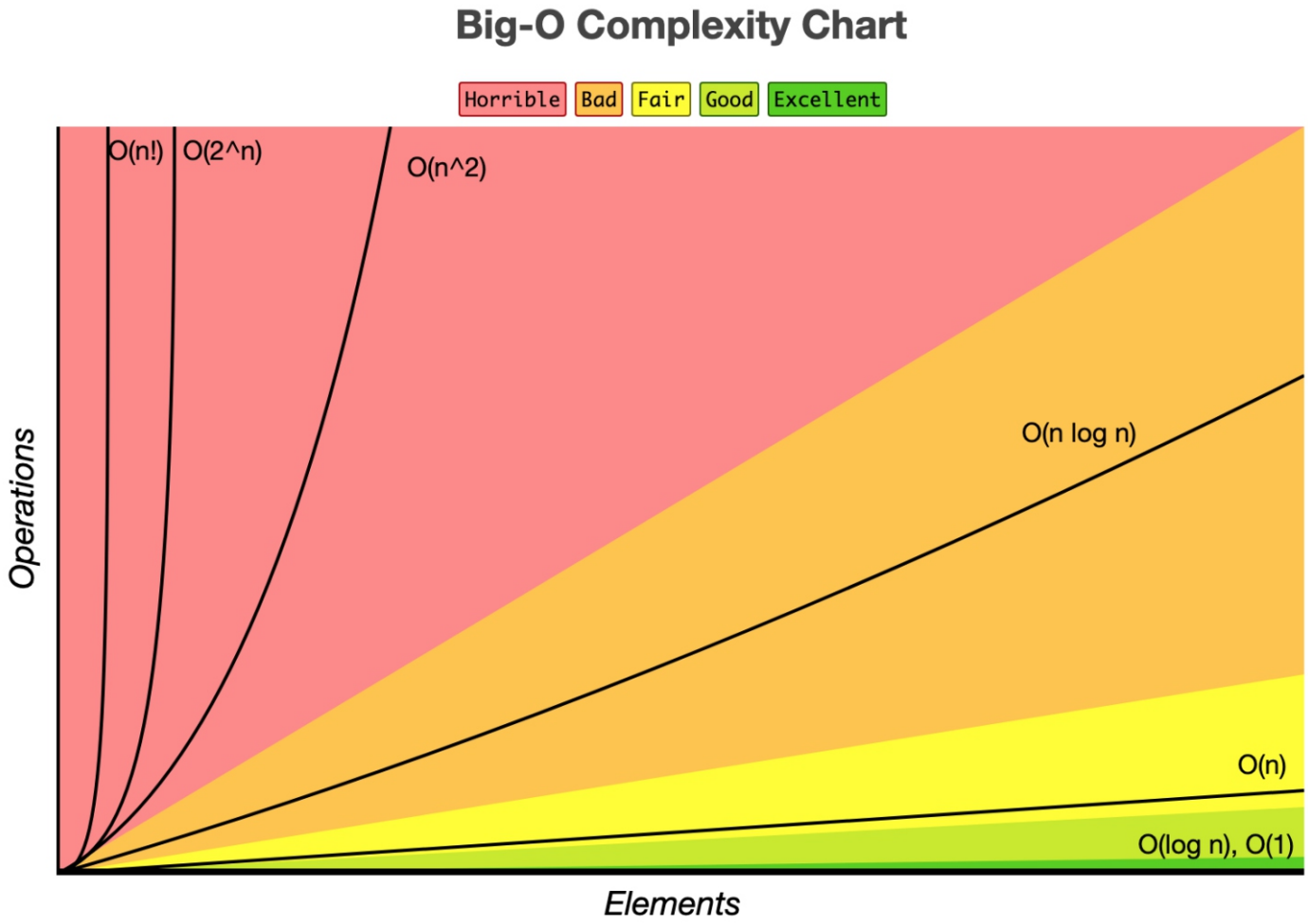

Big O
Arrays
Hash Tables
Single
Linked List
Doubly
Three
Binary Search
Binary Heap
Trie
Stack
Queue
Recursion
Graph's
Buble
Sorting
Selection
Insertion
Merge
Quick / Others
Searching
Linear
Binary
BFS
DFS

BIG O

Por definição o *Big O* é uma representação matemática que descreve o comportamento limitante de uma função que segue uma tendência geral em relação aos parâmetros que estão sendo manipulados.
Em outras palavras utilizamos o *Big O* para representar qual o tipo de notação da função, e para que seja possível, observamos o quão o nosso gráfico cresce a medida em que os parâmetros aumentam. Sendo que o gráfico simboliza o que chamamos de *time complexity*(número de operações que estão sendo realizadas)



O(n) - Linear

Para darmos introdução vamos começar com uma notação bem simples com uma função que calcula a soma de todos os números do número a ser informado.

```
function addUpTo(n) {
    let total = 0;
    for (let i = 1; i <= n; i++) {
      total += i;
    }
    return total;
}
```

por exercício de imaginação, tente idealizar quantas operações são realizadas na medida em que os parâmetros aumentam gradualmente.
Aora vamos ‘debugar’ a nossa função olhando diretamente para as operações.

constant

```
let total = 0;
(let i = 1;
```

= 2

loop

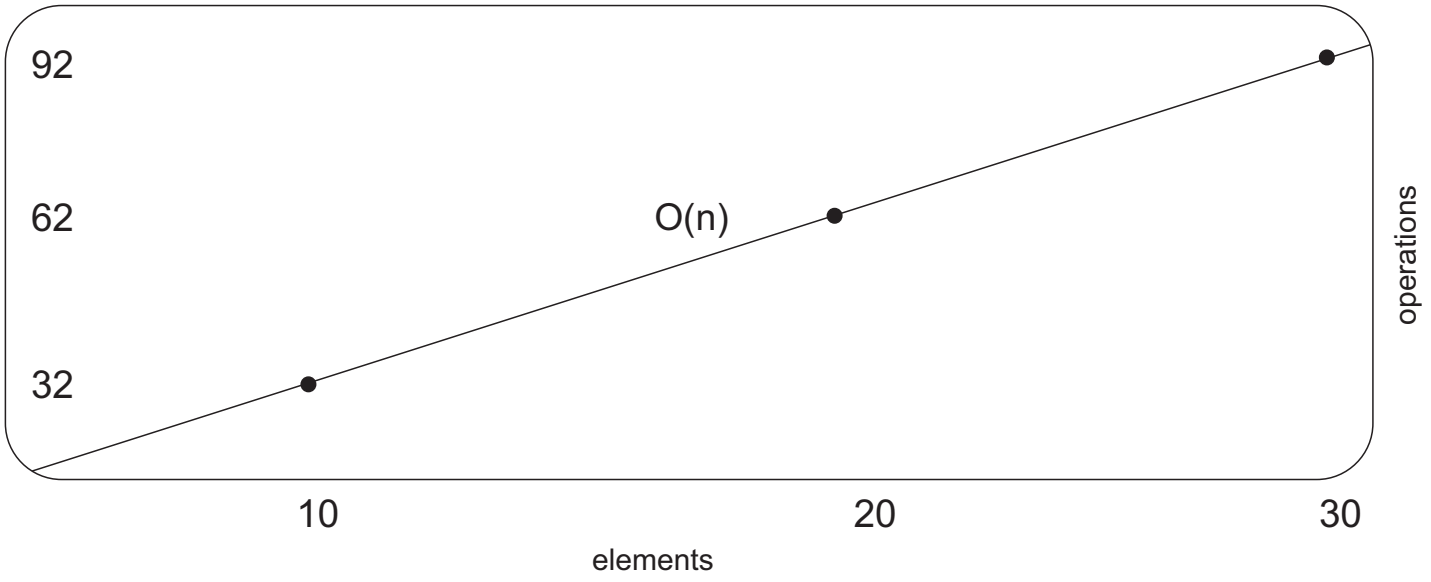
```
i <= n;
i++
total += i;
```

= n of parameters * 3

=> 2 + (parameters * 3)

De forma constante assim como mostra o primeiro quadro, sempre teremos operações fixas para inicializar as variáveis, e as três operações no segundo quadro representa o loop que terá o número de operações de acordo com o próprio parâmetro informado.
- *n* igual á 10, teríamos 2 + (10 * 3) = O(n32).
- *n* igual á 20, teríamos 2 + (20 * 3) = O(n62).
- *n* igual á 30, teríamos 2 + (20 * 3) = O(n92).

Logo em uma representação gráfica teríamos:



repare que a medida em que os nossos parâmetros aumentam, o nosso gráfico aumenta de forma linear. Então realmente em nosso caso não importa ter uma noção exata de qual número constante de ‘N’ a nossa função está sendo representada, o que realmente importa para nós é o ‘ponto central do desenho’ que neste caso simplesmente podemos chamar de O(n).

Para simplificar vamos usar outro exemplo com outra função onde o único intuito é realizar calculos sem qualquer fim.

```
function random(arr) {
  let n = 1;
  let x = 1;
  let sum = 0;

  for(let i = 0; i < arr.length; i++){
    n += arr[i];
    x += arr[i];
    sum += n*x;
  }

  return sum * 100;
}
random([5, 3, 2, 10, 20, 30, 40, 50, 22]);
```

observe que neste caso poderíamos contar unitariamente o número de operações a serem realizadas, como exemplo fazendo o uso de 10 entradas:

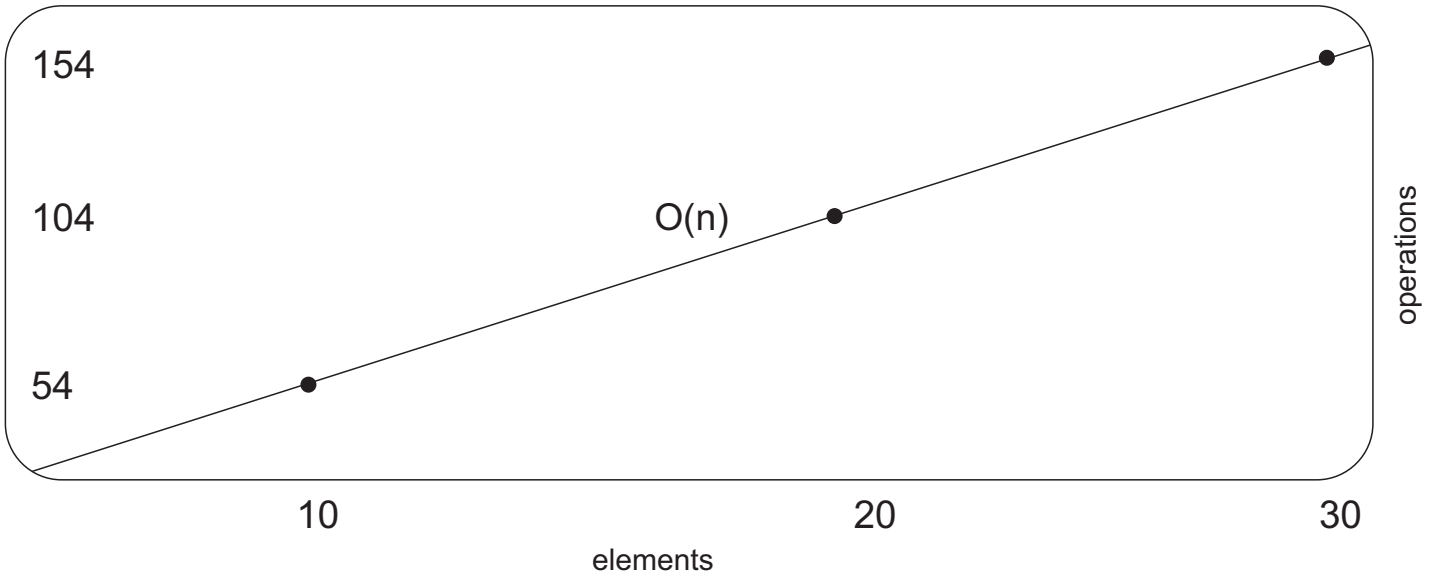
*- O(n + n + n + n + (nparameters * 5n)) => O(n4 + (10n * n5)) => O(n54)*

20 entradas:

*- O(n + n + n + n + (nparameters * 5)) => O(4n + (20 * 5)) => O(n104)*

30 entradas:

*- O(n + n + n + n + (nparameters * 5)) => O(4n + (20 * 5)) => O(n154)*



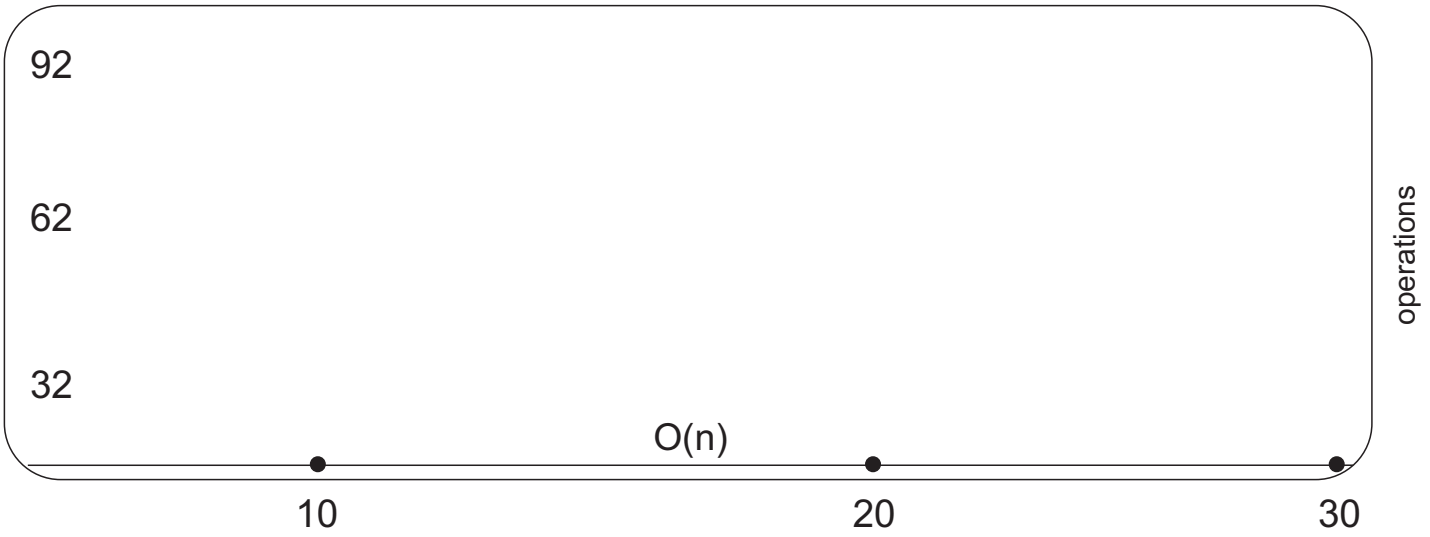
observe que realmente não importa as constantes, nós simplesmente podemos remove-las e dizer que a nossa função é um 'Big O(n)'. Isto faz com que não precisemos sair calculando o número exato de operações da nossa função, sendo o suficiente simplesmente entender como os nossos dados são trabalhados para que julguemos a forma como são representados.
No futuro conseguiremos diferenciar melhor os diferentes tipos de 'Big O's', principalmente quando trabalhamos com loop's aninhados. Mas no geral temos isto, é possível julgar o tipo simplesmente 'batendo o olho'.

O(1) - Constant

Agora vamos aprender uma representação de um ‘algoritímo perfeito’ chamado de ‘O(1)’, isto significa que realmente não importa o quão cresce a nossa entrada de parâmetros, o número de operações a serem realizadas sempre será o mesmo.
Usando como exemplo a mesma função anterior faremos uma refatoração de modo que tenhamos o mesmo resultado em um número constante de operações.

```
function addUpTo(n) {
  return n * (n + 1) / 2;
}
```

Ao executar a função acima, independente do quão grande seja o número a ser passado, somente executaremos uma operação singular que nos trará o mesmo resultado, logo teríamos o seguinte resultado no gráfico:



O(n^2) - Quadratic

Chegamos em mais um ‘Big O’, sendo este um dos mais ‘críticos’ que realmente queremos evitar, onde trata-se simplesmente da execução ao quadrada em um número de operações. Como exemplo, dado 10 entradas e o resultado final operante aproximado é de 100, temos um ‘O(n!)’. Este pode ser facilmente identificado quando temos loop’s aninhados, onde para cada iteração em um ‘looping pai’ temos o mesmo número de etapas em um ‘looping filho’ de forma sucessiva.

Como exemplo temos:

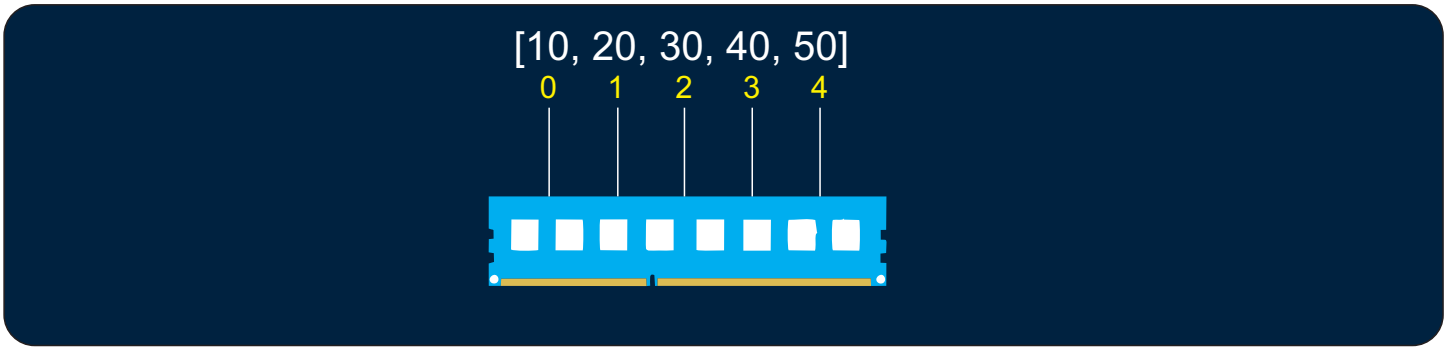
```
const boxes = ['A', 'B', 'C', 'D', 'E', 'F'];

function printAllBoxes(arr){
  for(let i = 0; i < arr.length; i++){
    for(let j = 0;  j < arr.length; j++){
      console.log(boxes[i], boxes[j])
    }
  }
}

printAllBoxes(boxes);
```

repare que para cada vez que o primeiro loop é percorrido temos um segundo loop que irá percorrer o mesmo número de iterações, nos levando a um algoritimo ao quadrado.

Ao observar o gráfico, é notório que a medida em que o número de elementos cresce, o número de operações cresce exponencialmente.

ARRAY

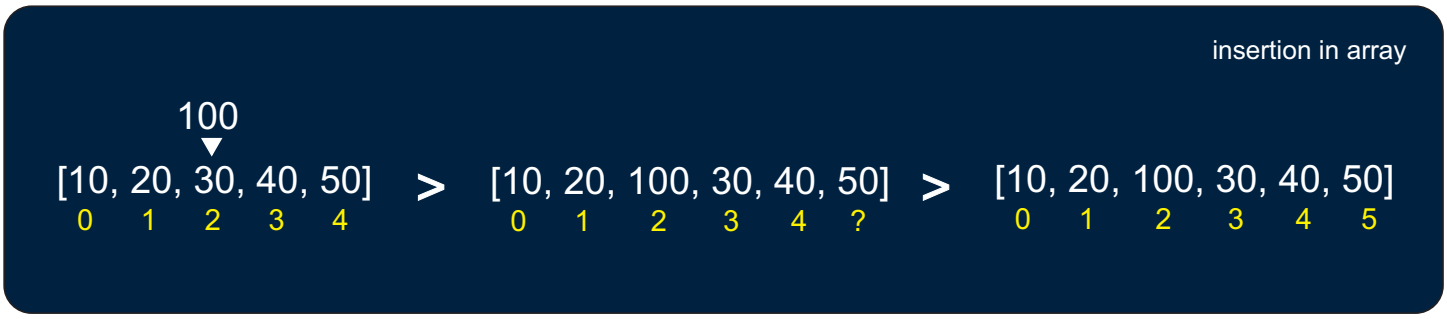
Array é uma estrutura de dados que armazena uma coleção de elementos de tal forma que cada elemento pode ser acessado diretamente por um índice ou uma chave. Sendo que uma das peculiaridades do Array é que cada elemento é posto em memória um ao lado do outro, e cada posição é representada por um índice.

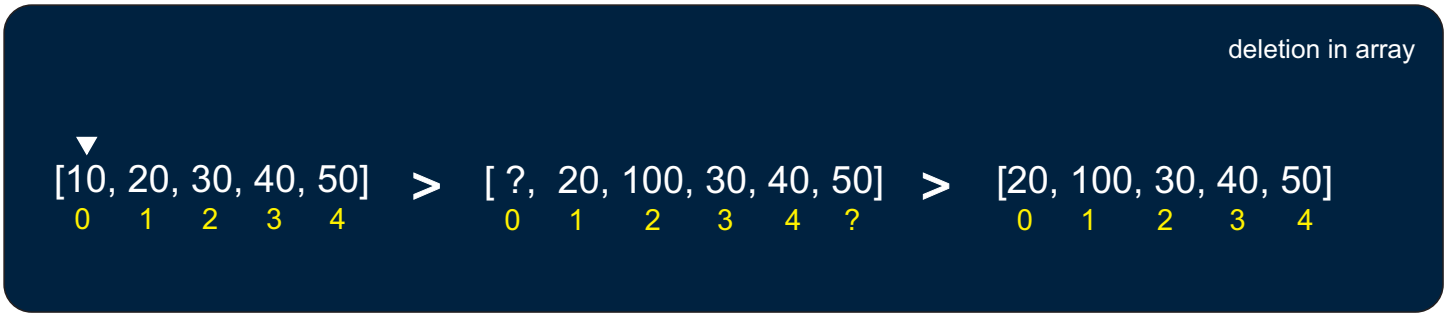


[ ]

Access	Search	Insertion	Deletion
O(1)	O(n)	O(n)	O(n)

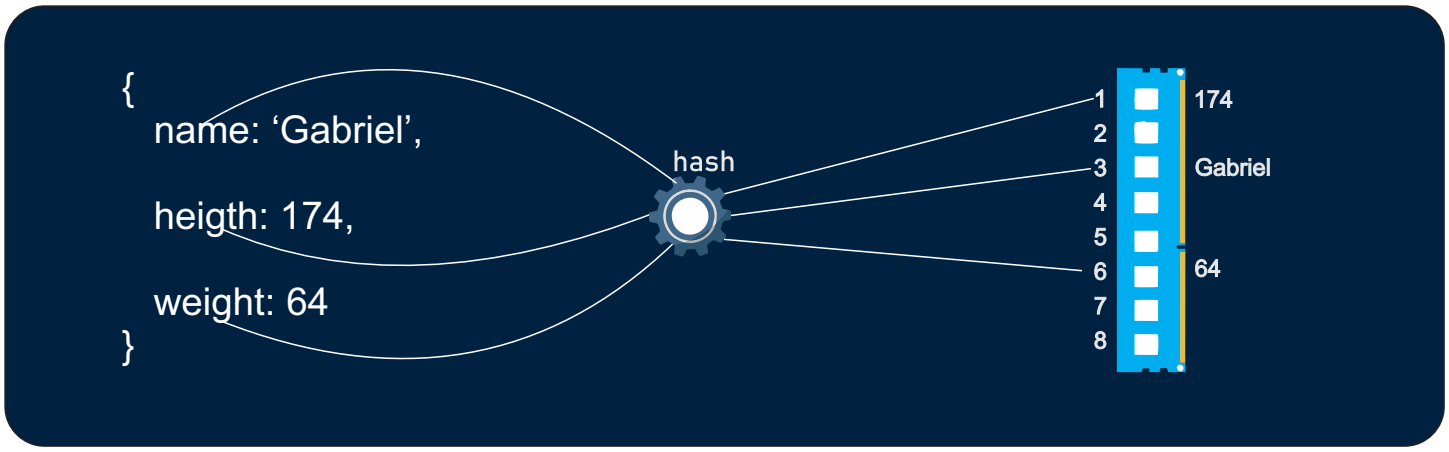
- O acesso a elementos em um array é extremamente rápido, já que ao acessar determinado índice, a estrutura de dados já guarda a referência em memória de onde aquele índice pode ser encontrado, o que torna uma operação constante.
- Buscas por igualdade em array's leva o 'tempo' de acordo com a largura do próprio do Array, já que precisamos acessar todos os elementos até encontrar o que de fato queremos, o que torna uma operação que vai de acordo com 'n' elementos.
- Inserções e deleções basicamente carregam em si o mesmo paradigma, pois caso queiramos inserir por exemplo um elemento no meio do array, precisaríamos realizar operações para reposicionar os demais índices depois deste elemento.



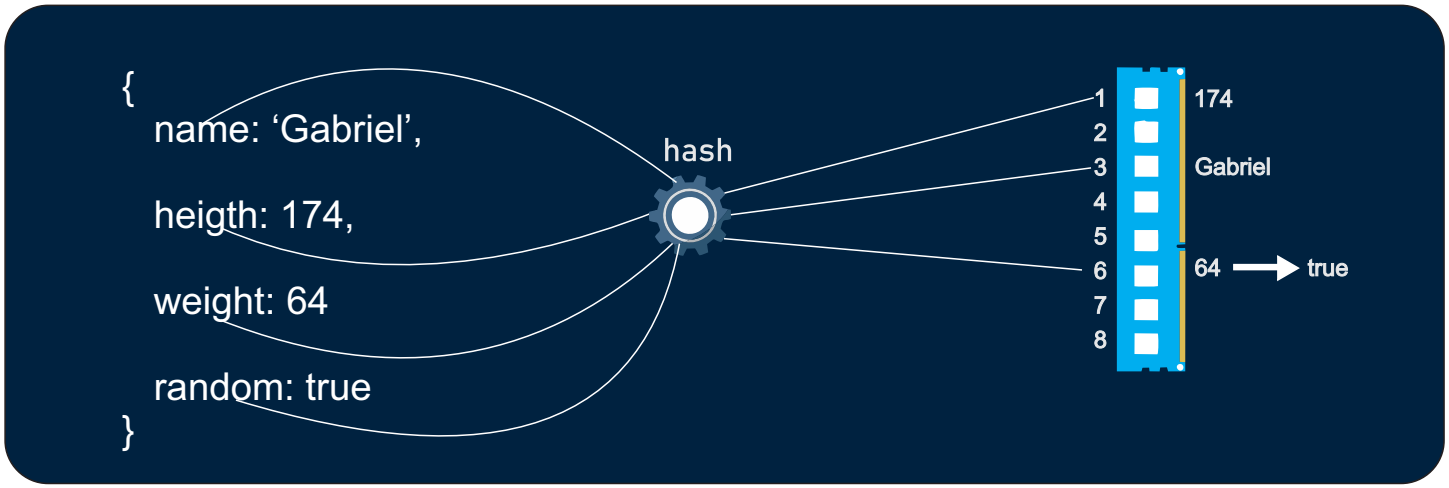


HASH TABLES

Hash tables (tabela de dispersão) é uma estrutura de dados que associa chaves e valores. O seu objetivo é a partir de uma chave simples fazer uma busca rápida e obter o valor esperado.



Observe que para cada chave é gerado um ‘hash’ referente a posição em memória do valor armazenado. E é possível que em determinados momentos podemos ter diferentes ‘hashs’ sendo calculados para uma mesma posição, o que ocasiona o que chamamos de ‘colisões’.



Nessas situações a estrutura de dados será responsável por criar a referência de apontamento para todos os valores ‘colididos’.

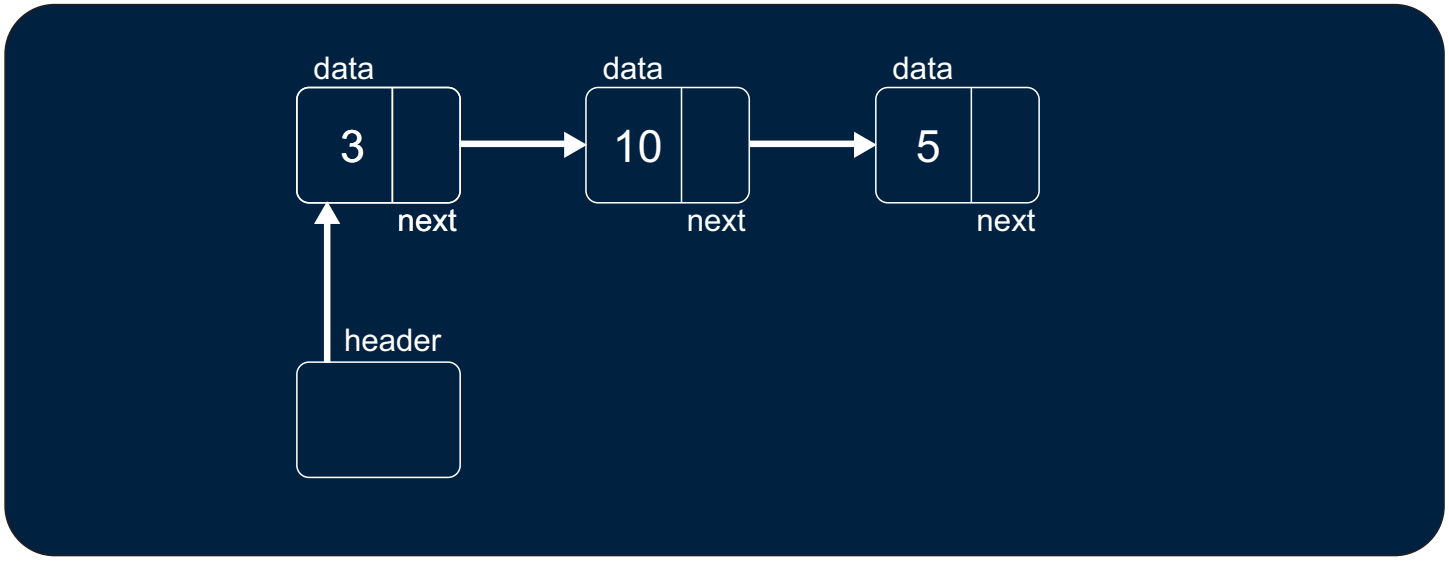
{ }

Access	Search	Insertion	Deletion
N/A	O(1)	O(1)	O(1)

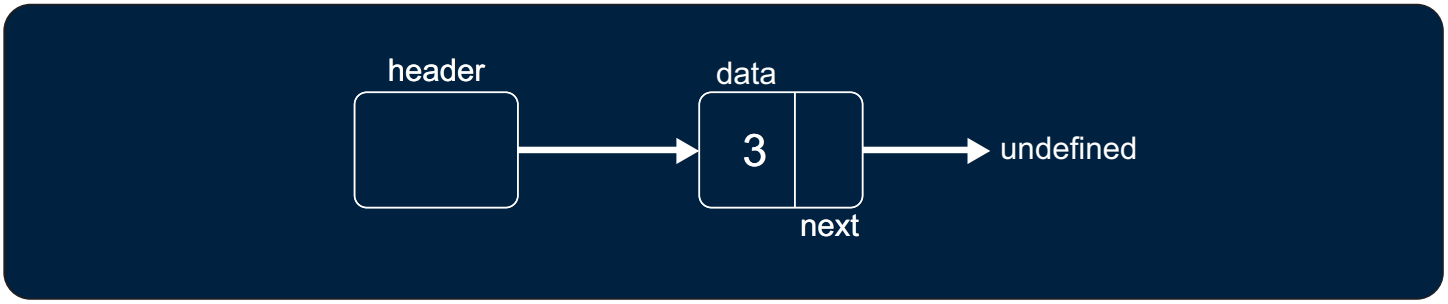
- É extremamente notório o quão dinâmico e rápido pode ser realizar uma busca por determinado elemento usando ‘hash tables’, visto que simplesmente podemos acessar diretamente o valor através da referência pela chave que obteremos o resultado esperado.
- Vale o mesmo para inserções e deleções visto que a operação de gerar um ‘hash’ sempre será uma operação constante visto que não há a necessidade de reposicionar os demais elementos (como vimos anteriormente com os array’s)

LINKED LIST

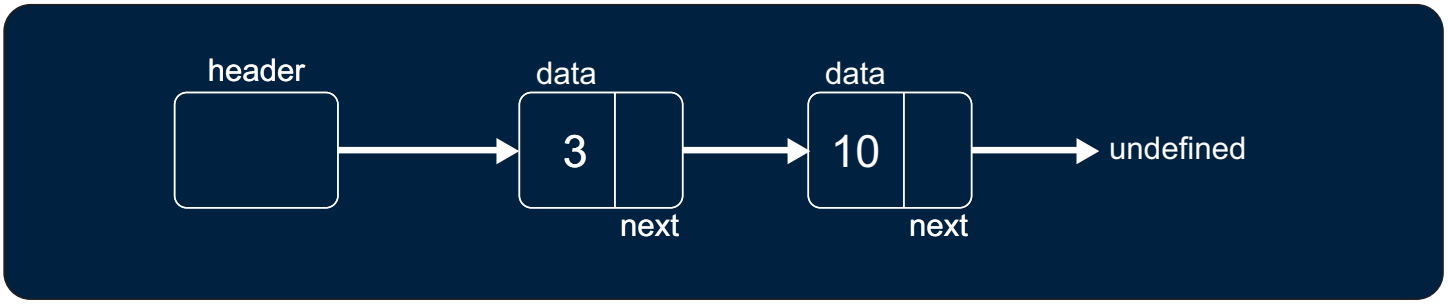
Uma lista ligada é uma estrutura de dados que é composta por várias células interligadas através de ponteiros, ou seja, cada nó aponta para o endereço em memória do próxima nó e assim sucessivamente desse modo, as células da estrutura não precisam estar em posições contíguas da memória.



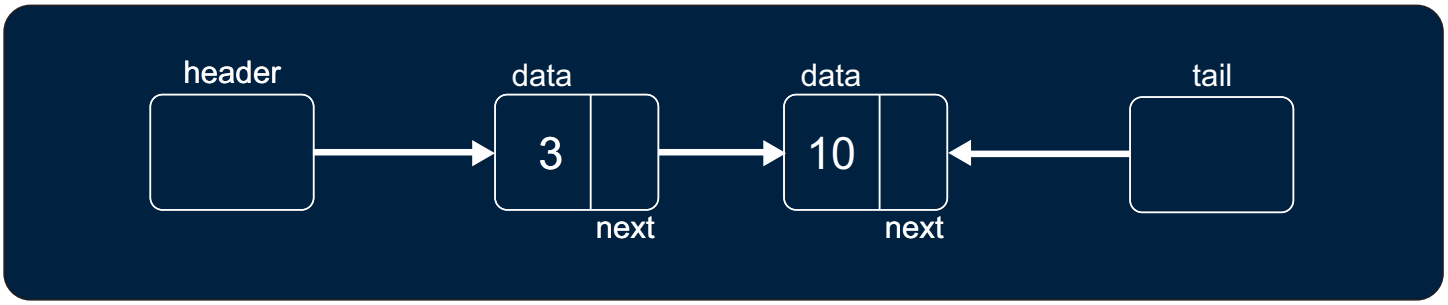
A estrutura de criação para uma lista ligada é dada da seguinte forma: a fila é inicializada com um elemento definido, em seguida criamos uma posição em memória cujo chamamos de header para referênciar o primeiro elemento, neste caso o elemento inicializado.



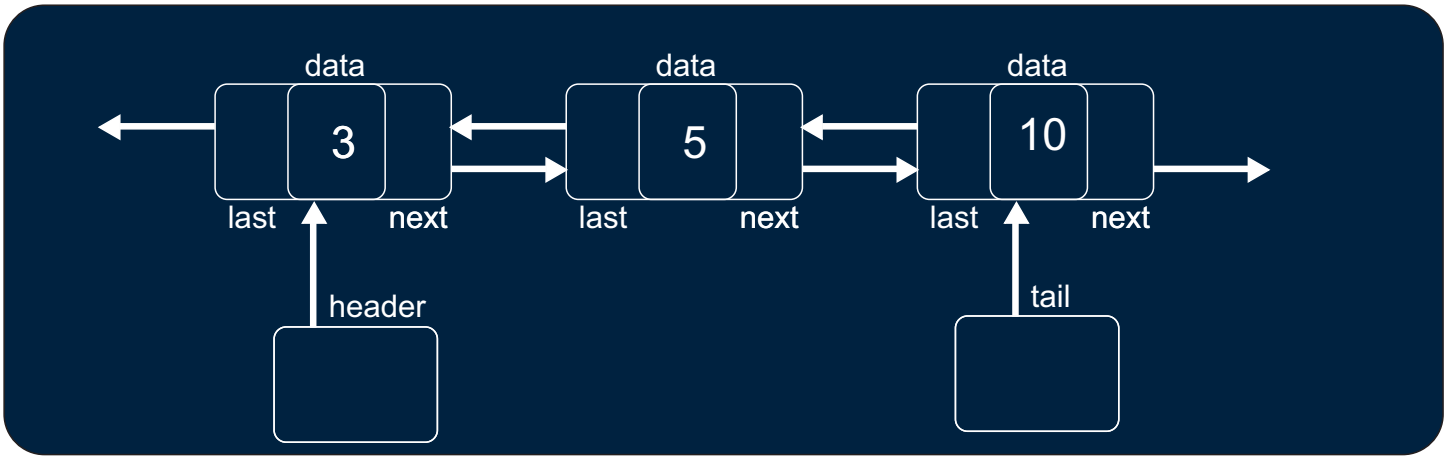
E a medida em que vamos criando elementos vamos referenciando os apontamentos de acordo com ordem.



Também podemos criar listas ligadas referenciando a ‘cabeça’ e a ‘cauda’ do nosso esqueleto.



Também podemos ter listas duplamente ligadas, onde um nó aponta para o próximo da ‘fila’ e respectivamente para o anterior.



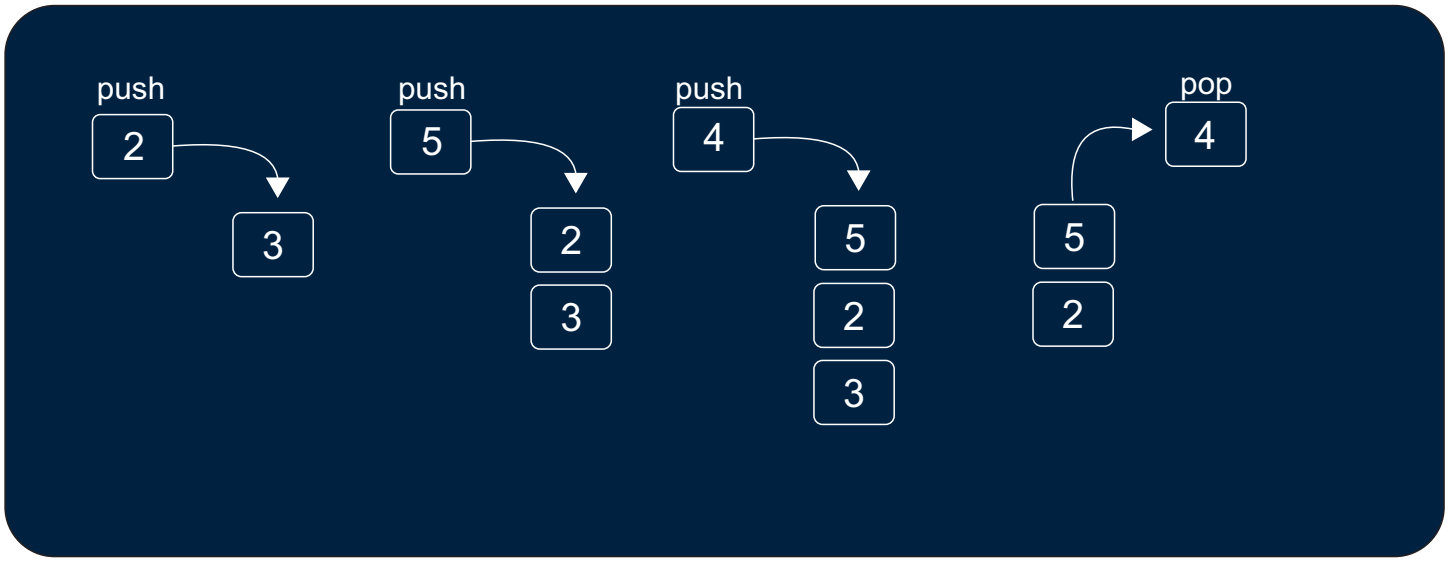
a vantagem de termos este tipo de lista é que conseguimos percorrer os elementos de forma transversal, o que torna certas situações mais dinâmicas.

Access	Search	Insertion	Deletion
Θ(n)	O(n)	O(1)	O(1)
		head Θ(1)	head Θ(1)
		tail Θ(1)	tail Θ(1)

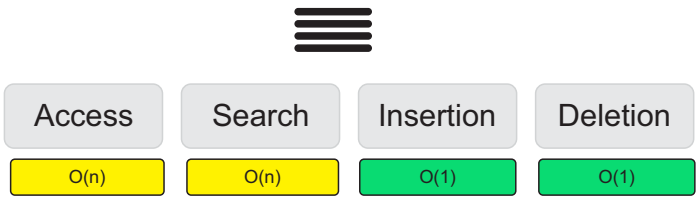
- Inserções e deleções considerando elementos no final ou no início da lista são operações constantes considerando que simplesmente devemos reposicionar o apontamento dos nós que estão em volta do elemento a ser deletado ou inserido.
- Diferentemente dos arrays em que temos a referência da posição em memória através de índices para acessar os elementos, na lista ligada temos que percorrer o nosso esqueleto de modo que encontremos o valor interessado, valendo mesmo para as buscas.

STACK'S

Stac (pilhas) são estrutura de dados para agrupar elementos base no princípio *Last In First Out,* ou seja o último que entra é o primeiro que sai.

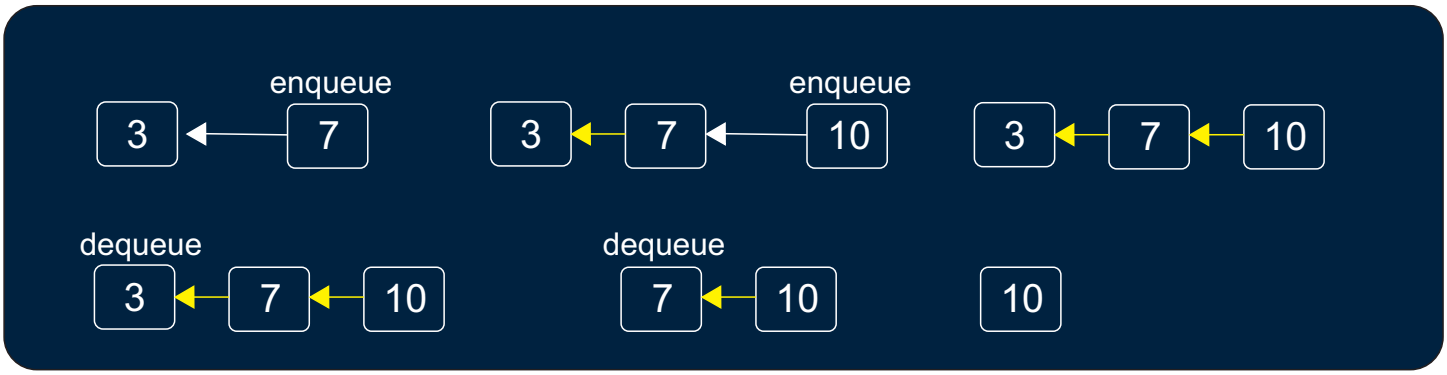


Pilhas podem facilmente serem implementadas através de Linked list's (listas ligadas) ou Arrays. Sendo a implementação através de array's mais simples porém com recurso limitado de memória (caso a linguagem a ser utilizada não exista um auto-redimensionamento)



QUEUE'S

Queu's (filas) é uma estrutura de dados sob o princípio de *First In First Out*, ou seja o primeiro a entrar será o primeiro a sair.



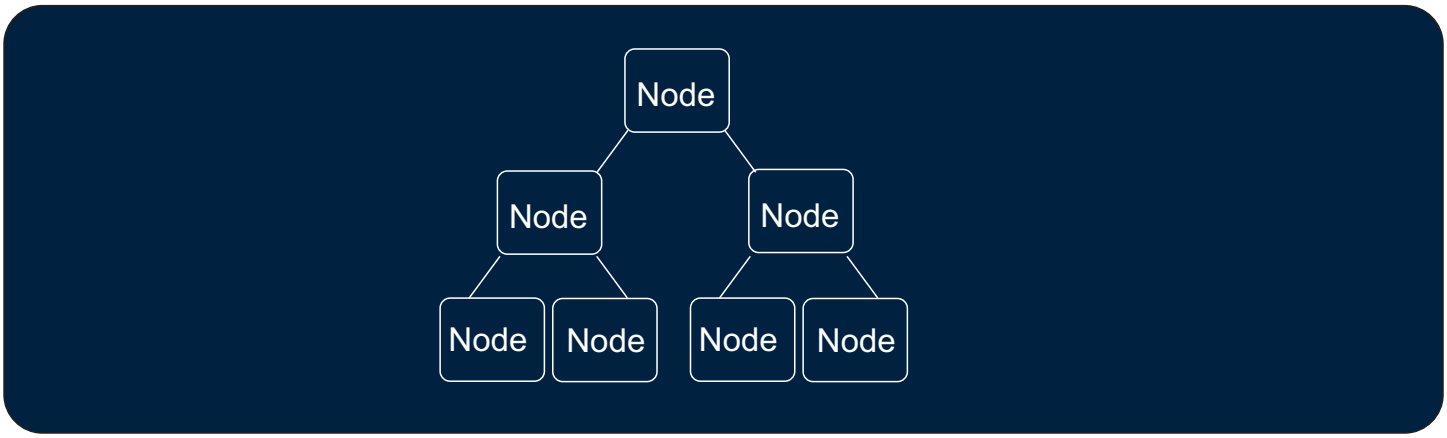
Como pode observar, filas obrigatoriamente devem ser implementadas através listas ligadas. Pois caso sejam implementadas através de array's, perderíamos o poder de operação constante ao remover do início, já que teríamos que reposicionar toda a fila toda vez.

TREE'S

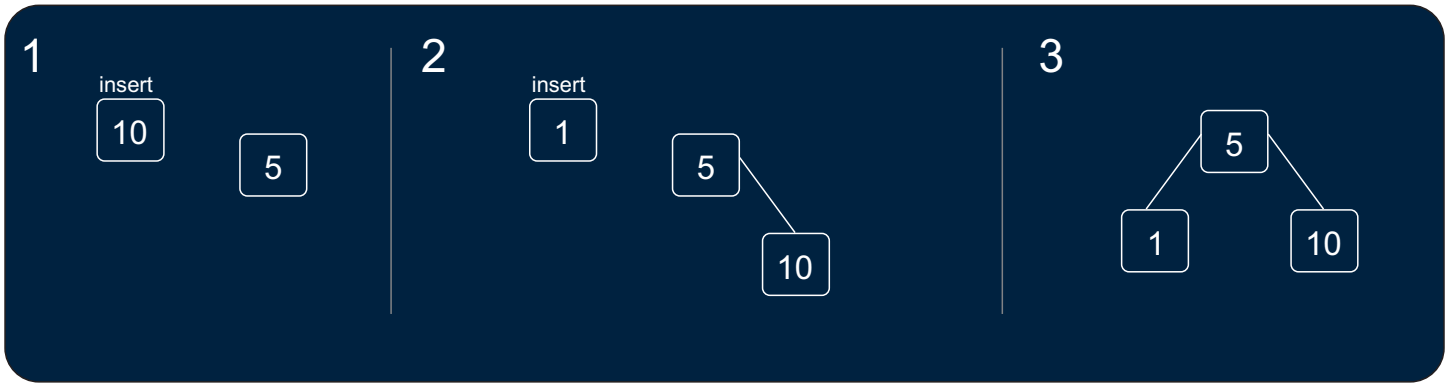
Trees (árvores) é um tipo de estrutura de dados em formato de árvore
onde os elementos estão conectados assim como listas ligadas(sendo listas ligadas também um tipo de estrutura de árvores) sob a diferença que temos algumas regras e referências definidas. Existem diversos tipos de árvores, comecemos falando a respeito sobre Binary Sarch (Árvore Binária de Busca)

BINARY SEARCH

Arvore binária de busca é um tipo de árvore onde um nó pode ter somente nenhum, um ou no máximo dois filhos e todos os elementos são alocadas a direita ou a esquerda sob o nó a ser inserido.



Sendo a regra para para inserção é verificar se o nó a ser inserido é maior do quê o nó atual, se sim novo nó é inserido a direita, se não inserido a esquerda.

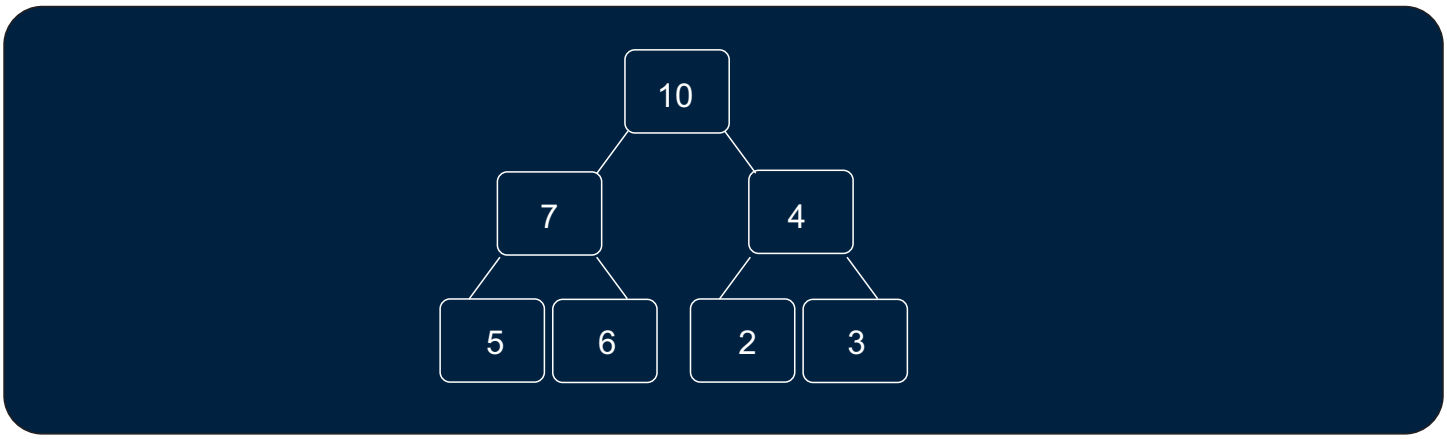


Access	Search	Insertion	Deletion
O(log n)	O(log n)	O(log n)	O(log n)

Agora vamos falar sobre o quão poderoso pode ser usar árvores binárias de buscas, como já falamos anteriormente, lista ligadas nos fornecem uma memória dinâmica sem limite previamente definido, além disso atravessar por ela sempre será uma notação 'quase constante', isso porquê os nossos elementos seguem um padrão de inserção e faz com que sempre encontremos os nossos nó's em determinada região a árvore. Pense nisso como uma lista telefônica, você em vez de percorrer toda a lista, entrará em atalhos que condizem com as inicias que você procura. Certamente em determinados momentos, no pior dos casos podemos ter uma lista desbalanceada onde os nós se conectam em um padrão muito parecido de listas ligadas, o que torna qualquer tipo de busca uma operação O(n). É por conta disto que existem alguns algoritimos de balanceamento como AVL ou Red/BLACK

BINARY HEAP

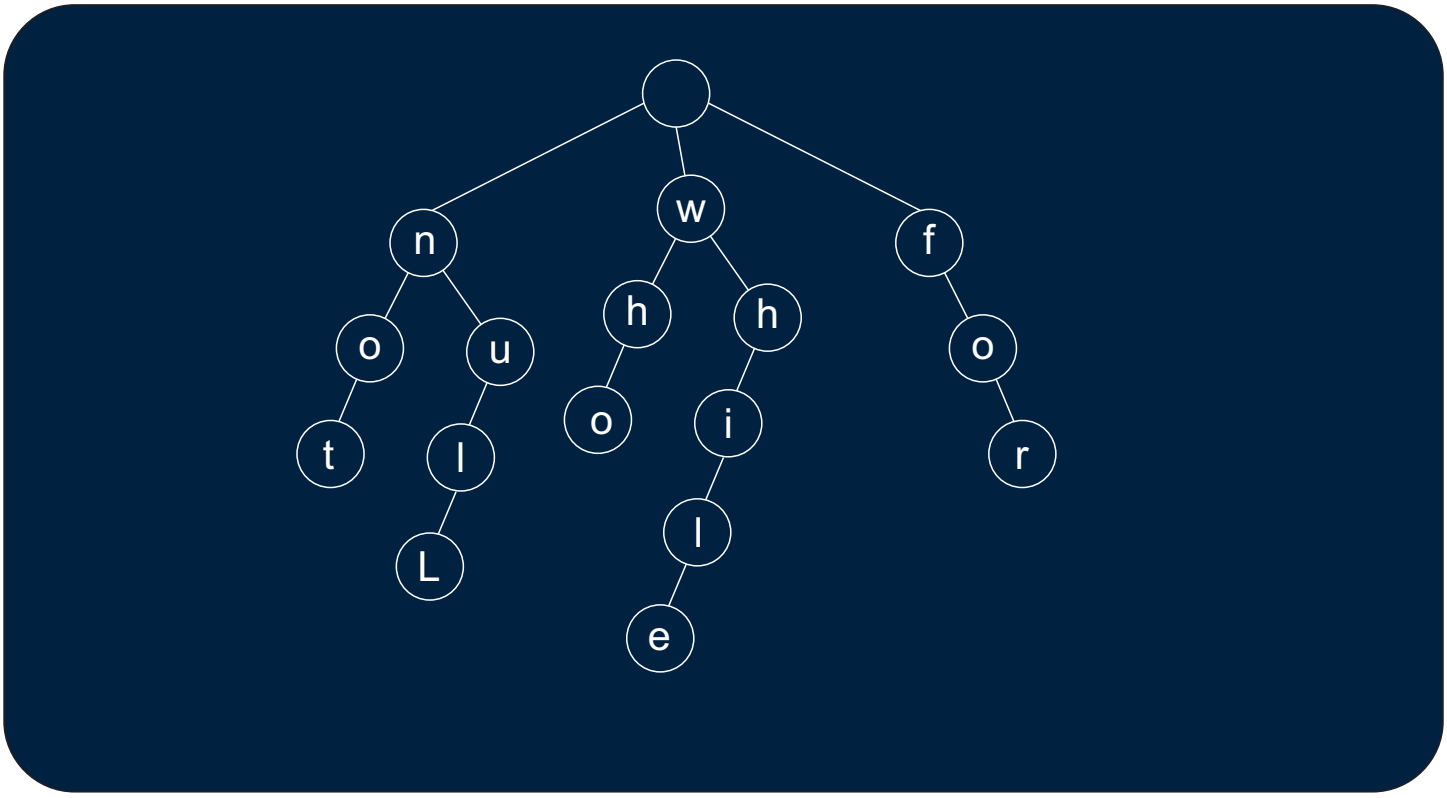
Árvore Binária de Pilha é um tipo de árvore de elementos muito parecido com a árvore binária de busca que vimos, sendo que neste caso o nó com valor maior sempre estará acima dos níveis dos demais nós sejam eles filhos ou não.



Este tipo de estrutura de dados é muito utilizado em algoritimos de sorting devido a sua alta capacidade de ordenar elementos e sua acessibilidade bastante flexível.

TRIE

Por fim chegamos em mais uma árvore, sendo a sua definição por nome muito parecida com o próprio tipo de estrutura de dados que estamos trabalhando: tree.
Essa estrutura pode ser usada para armazenar valores de forma associativa em que as chaves são normalmente cadeias de caracteres.



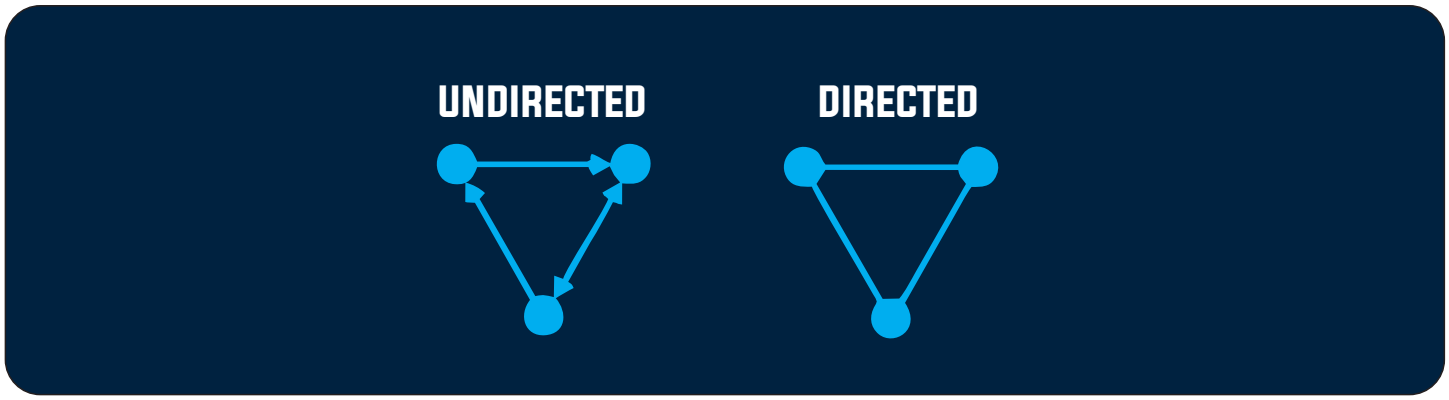
Este tipo de estrutura de dados pode ser muito útil em ordenação de um conjunto de valores como uma string por exemplo.

GRAPH'S

Graph's por representação é uma estrutura de dados muito útil para adaptar conexões do "mundo real" em forma de dados. Então é certo quando dizemos que as árvores são um tipo de grafo. Existem milhares tipos diferentes de grafos, mas podemos representa-los através de algumas características.

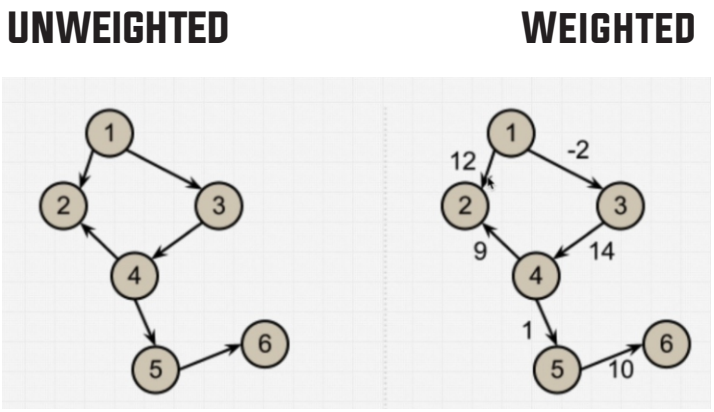
UNDIRECTED / DIRECTED

Undirected é quando os nós estão conectados de forma bidirecional, ou seja, conseguimos ir e voltar a partir de dois nós que estão conectados desta forma.
Já o Directed é quando há somente um caminho entre um nó e outro, não sendo possível voltar o caminho feito a partir de um nó.

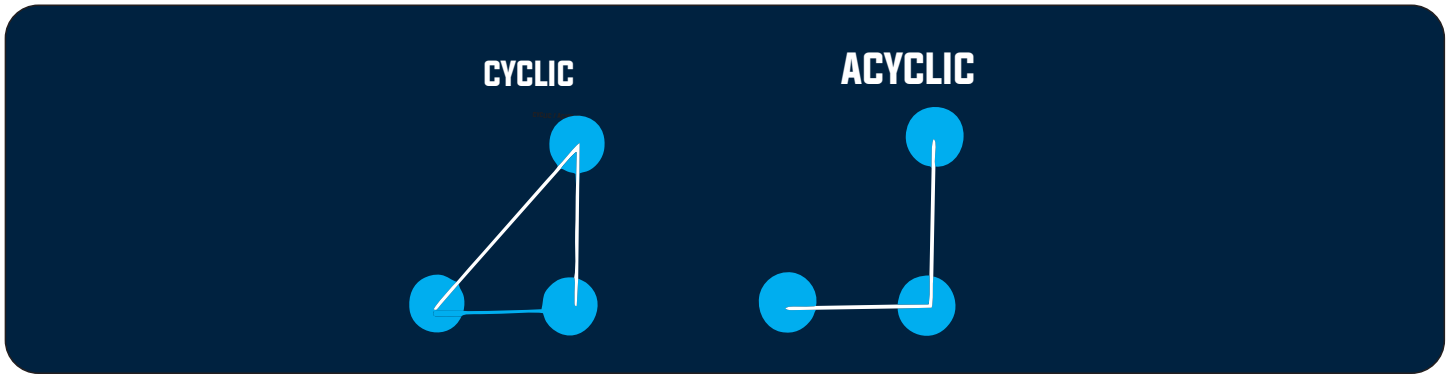


UNWEIGHTED / WEIGHTED

Weighted é quando o valor do nó é armazenado no laço (edge) de conexão com o próximo nó.
Unweighted é quando o valor é armazenado no próprio nó.



Um grafo Cyclic é quando conseguimos percorrer em volta de todo nosso grafo através de caminhos opostos, algo como andar em círculos em volta de uma estrada circular. Já o Acyclic é o oposto, quando não temos essa conexão direta por volta das conexões.



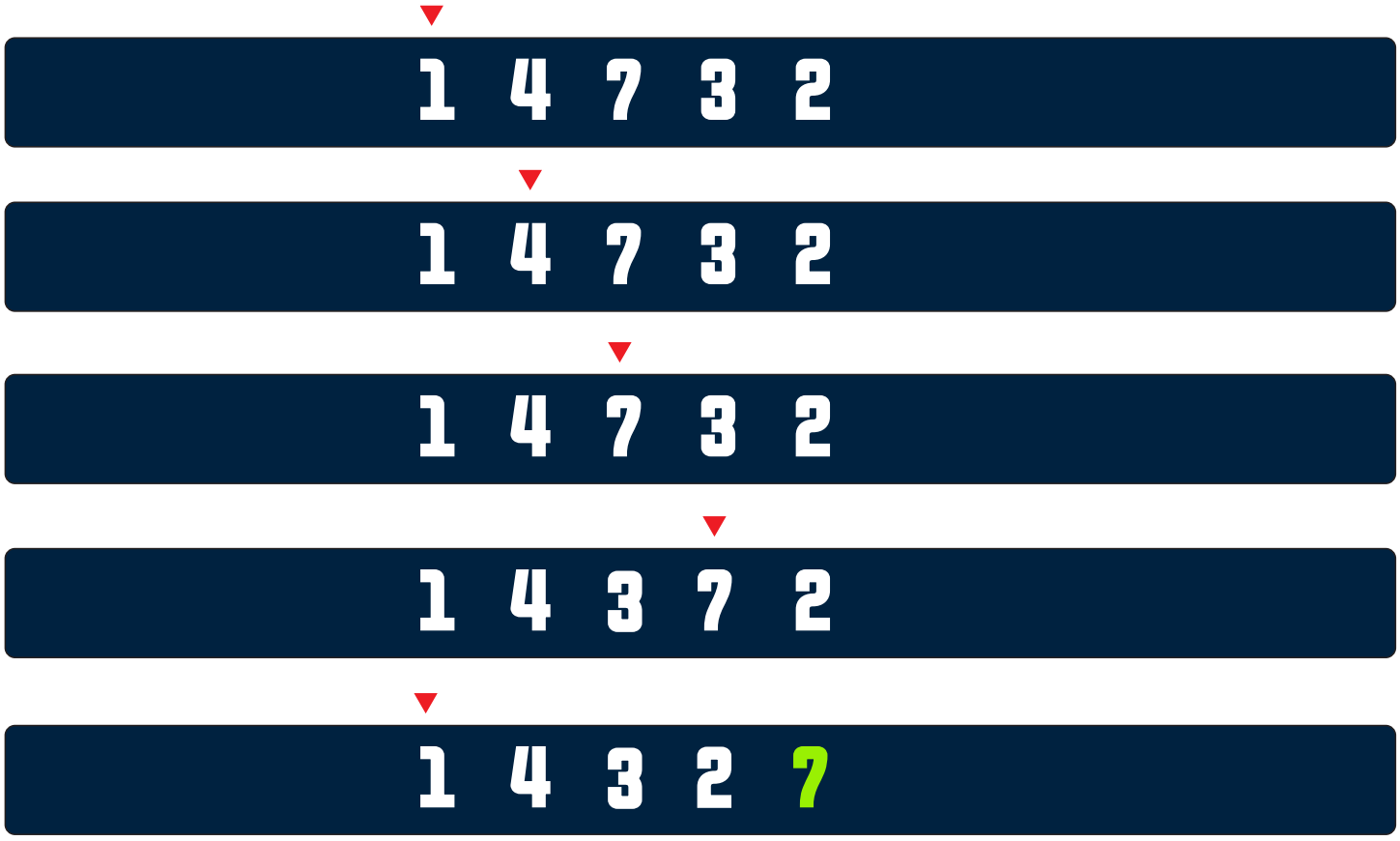
SORTING

Algoritmo de ordenação em ciência da computação é um algoritmo, de manipulação de dados, que coloca os elementos de uma dada sequência em uma certa ordem. Existem diversos algoritimos de ordenação, sendo cada um mais bem aplicados em abordagens específicas.

BUBBLE

O algoritimo bubble sort é um dos mais simples, a ideia é percorrer o array diversas vezes onde cada interação levará o maior valor ao último índice. Vamos acompanhar o fluxo com o seguinte conjunto de dados.

A prática base é a seguinte: comecemos iterando a partir do primeiro elemento já verificando se o próximo elemento é menor do que o valor atual, se não for deixamos as posições como estão, porém se for, invertemos ambos e continuamos iterando, assim fazemos sucessivamente.

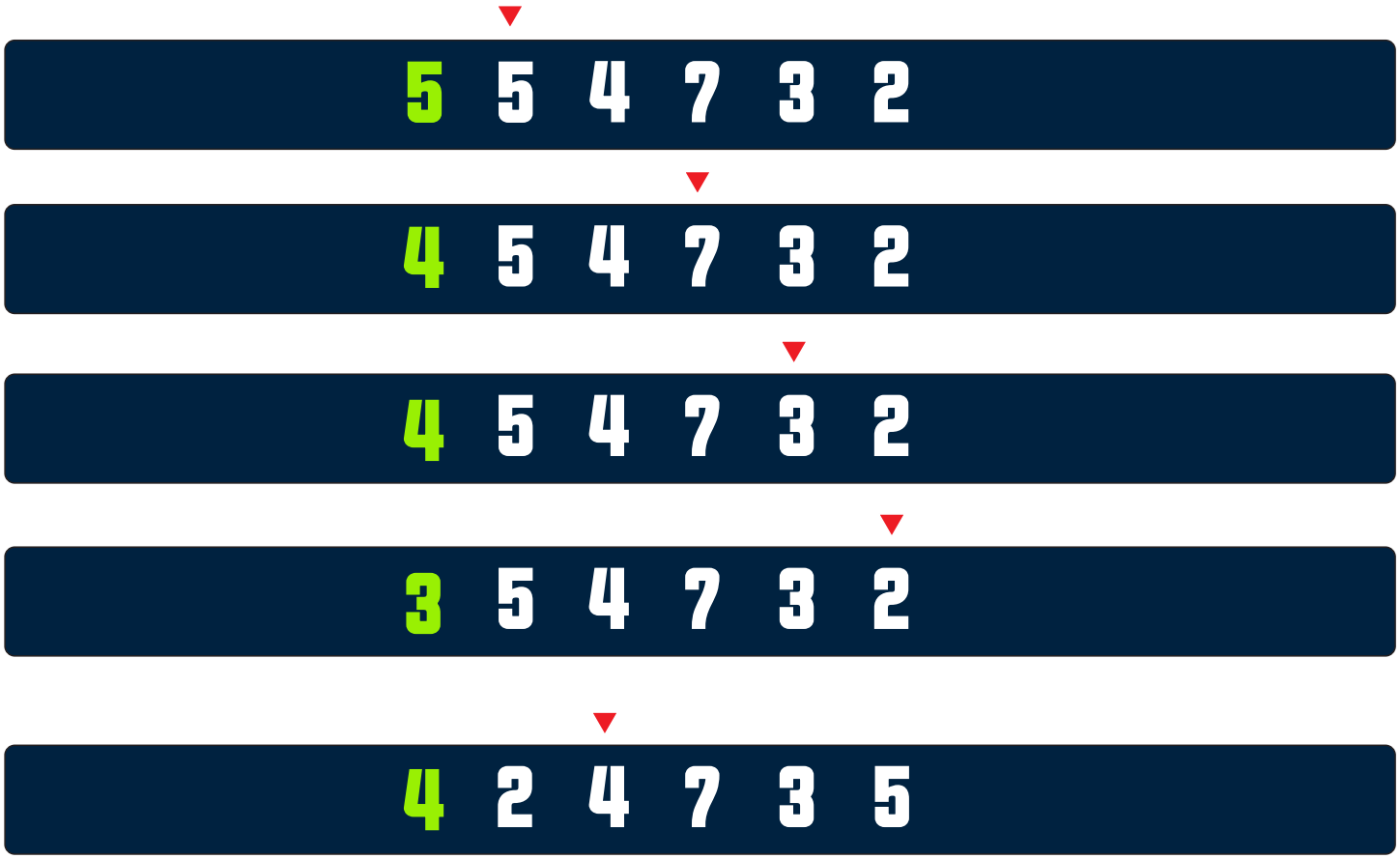


Agora que no primeiro looping já movemos o maior elemento ao extremo do array continuamos a realizar as mesmas etapas iterando até N-1 pelo fato de neste primeiro momento não precisar verificar o último índice.

SELECTION

O algoritimo ‘selection’ é um pouco parecido com o ‘bubble’ que vimos anteriormente, sendo a diferença é que para cada iteração nós guardmos o menor elemento de todos em uma posição em memória para que seja enviado ao topo e assim sucessivamente.

Já começamos inicializando o primeiro elemento como o menor, e verificamos se a próxima posição é menor do que o menor atual, se sim o menor atual passa a ser o próximo elemento, se não o menor permanece intacto e continuamos iterando.



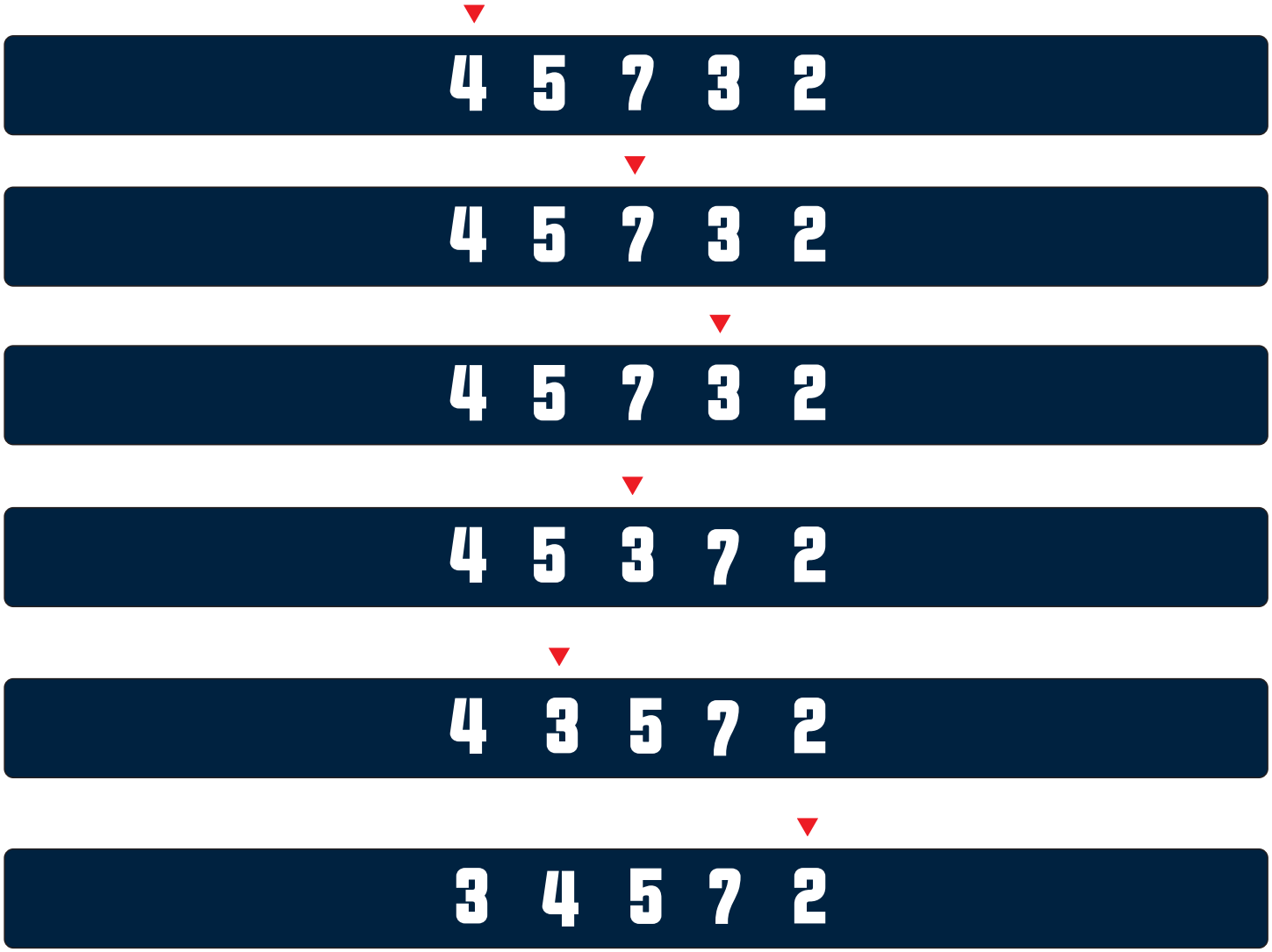
continua....

enfim falamos um pouco sobre os dois mais simples algoritimos de ordenação que em larga escala jamais serão usados pelo simples fato de existir algoritimos bem mais complexos e eficientes. Agora falaremos sobre um método de ordenação já um pouco mais aplicável.

INSERTION

O algoritimo insertion sort também segue um conceito simples, a idéia é iterar em cima de cada elemento, e para cada iteração olhar para os elementos a esquerda até achar a sua posição de encaixe, sendo o caminho de etapas dada da seguinte forma:

Comecemos já olhando para segunda posição e verificando se o primeiro elemento a esquerda é maior, se sim nós invertemos o número atual de posição e seguimos com a mesma verificação (para o mesmo número) sequencialmente até chegar ao último índice ou até encontrar um número menor.

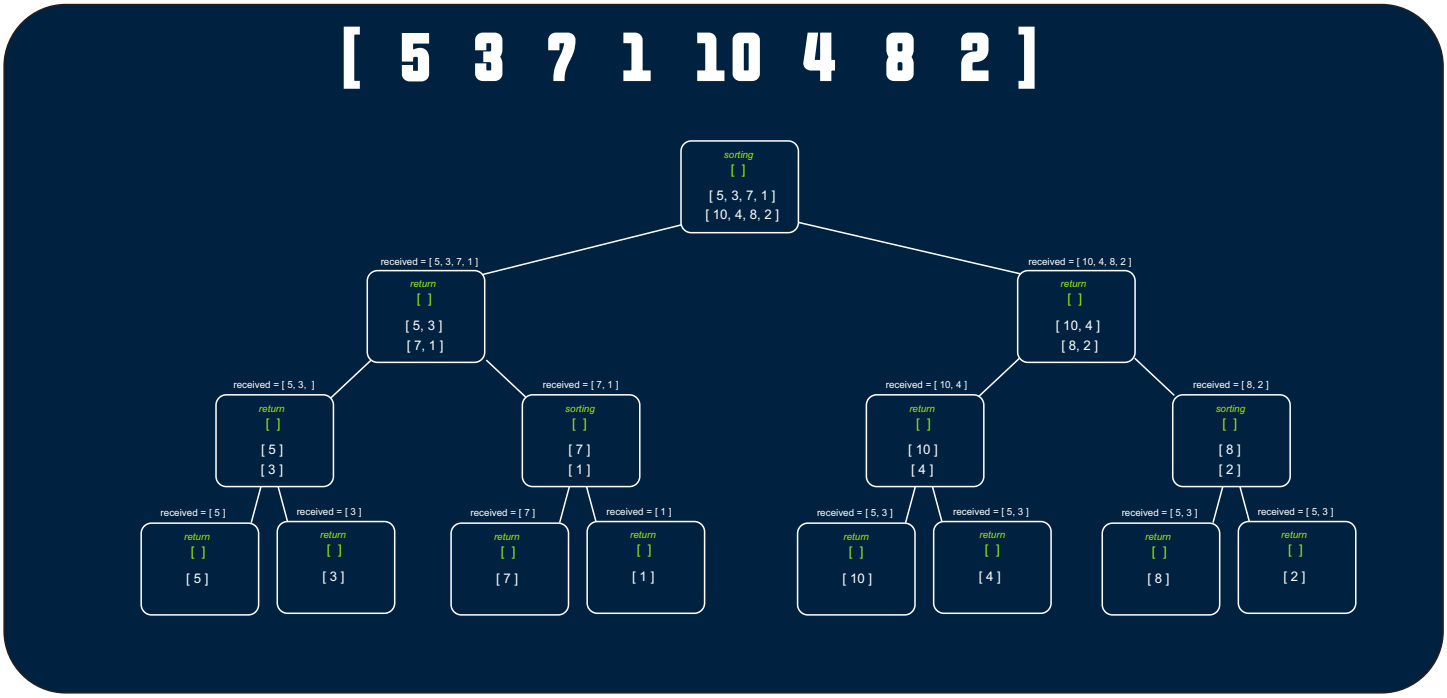


continua....

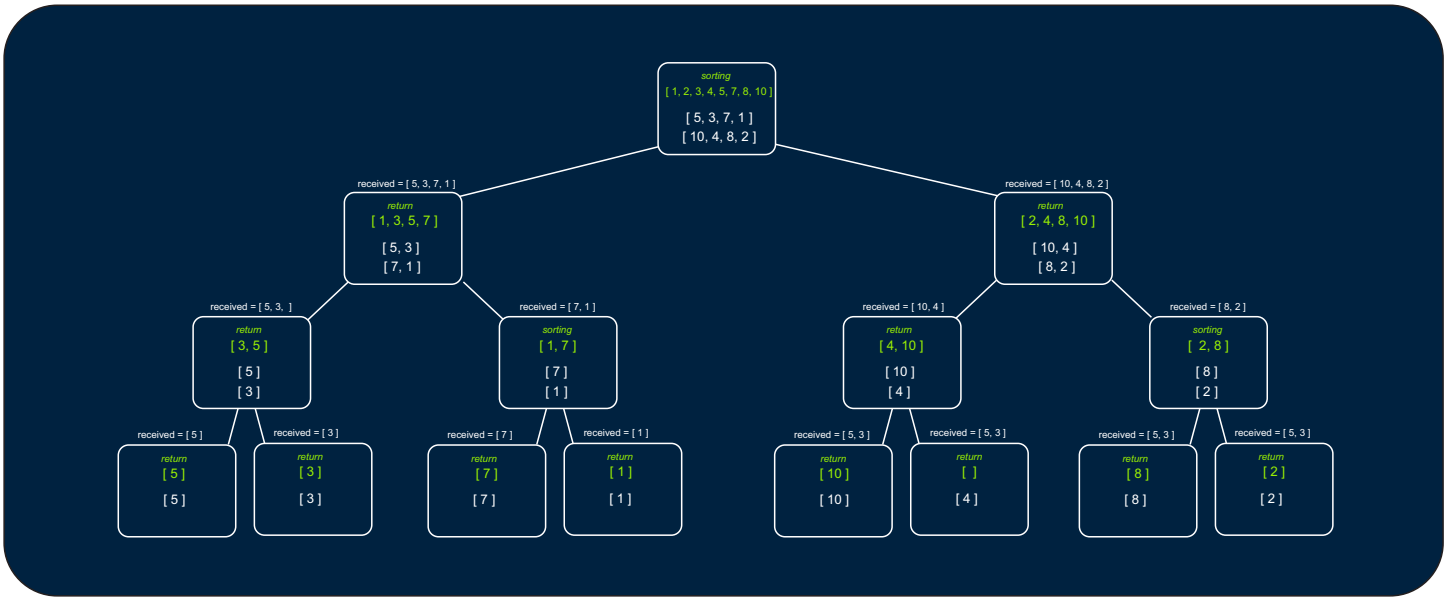
este algoritimo geralmente é indicado para elementos que estão quase ordenados, ou completamente ordenados.

MERGE

Finalmente chegamos em um dos algoritimos de ordenação mais eficiente e em termos de complexidade de tempo. Este consiste no simples conceito de dividir e conquistar, a divisão é simplesmente quebrar os dados em subpeças dentro de uma estrutura, e a conquista é resolução de cada subpeça com as demais subpeças presentes no todo conectadas através de funções recursivas.
Antes de exemplificar como o fluxo ocorre, tenha em mente sempre que as etapas em si seguem um fluxo bilateral, onde a primeira etapa é quebrar os pedaços de dados (dividir) e ordena-los de acordo com suas conexões (conquistar)
Considerando o seguinte array, vamos realizar a nossa primeira etapa que é dividir:

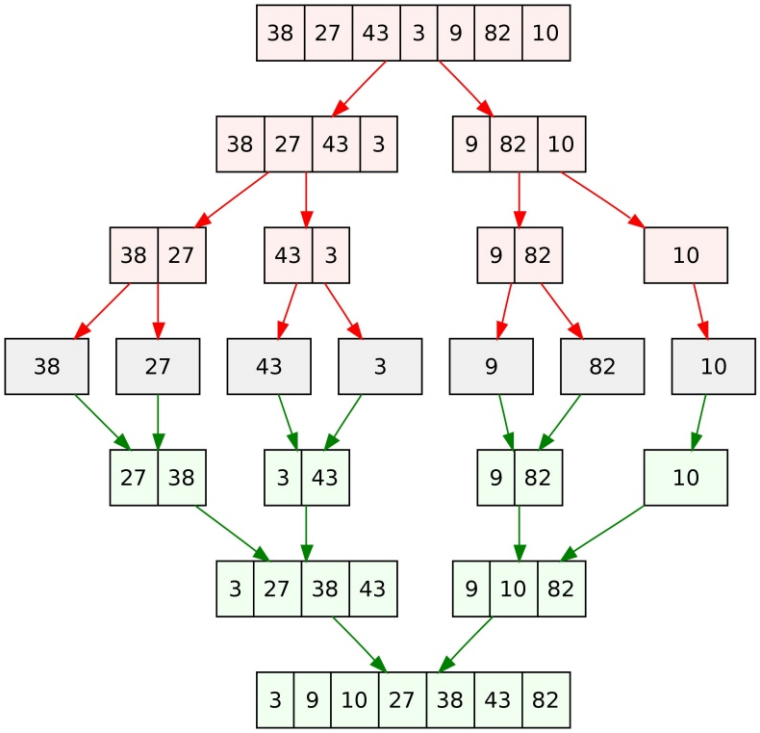


Repare que neste primeiro momento nós subdivimos nossos dados em duas partes para cada estrutura, para que essas partes criem outras sub estruturas até a profundidade. Cada estrutura está anexada a um algoritimo de sorting, em nosso caso representado pelo array de retorno. Agora a próxima etapa é enviar o conjunto ordenado ao pai de forma recursiva.



QUICK SORT

O algoritimo de ordenação quick sort também segue um conceito muito parecido presente no merge sort de divisão e conquista, sendo que a diferença é que o ponto de ordenação leva em consideração um pivor obtido aleatóriamente para ordenar os valores menores a esquerda e os maiores a direita e assim sucessivamente.



A vantagem de utilizar este método é que o nosso ‘space complexity’ não aumenta tanto na proporção em que os dados crescem. Já a desvantagem é que quando os dados não estão balanceados acabamos operando em um tempo ao quadrado. A conclusão é que se você tem a possibilidade de gerenciar o máximo o possível a escolha do seu pivor, este método de ordenação será bem performado principalmente em termos de memória, caso contrário será criada uma redundância e o poder de nível de operações será perdido.

RADIX SORT

O Radix sort é um algoritmo de ordenação rápido e estável que pode ser usado para ordenar itens que estão identificados por chaves únicas. Cada chave é uma cadeia de caracteres ou número, e o radix sort ordena estas chaves em qualquer ordem relacionada com a lexicografia.
Na ciência da computação, radix sort é um algoritmo de ordenação que ordena inteiros processando dígitos individuais. Como os inteiros podem representar strings compostas de caracteres (como nomes ou datas) e pontos flutuantes especialmente formatados, radix sort não é limitado somente a inteiros.

COUTING SORT

Counting sort é um algoritmo de ordenação estável cuja complexidade é O(n). As chaves podem tomar valores entre 0 e M-1. Se existirem k0 chaves com valor 0, então ocupam as primeiras k0 posições do vetor final: de 0 a k0-1.

A ideia básica do counting sort é determinar, para cada entrada x, o número de elementos menor que x. Essa informação pode ser usada para colocar o elemento x diretamente em sua posição no array de saída. Por exemplo, se há 17 elementos menores que x, então x pertence a posição 18. Esse esquema deve ser ligeiramente modificado quando houver vários elementos com o mesmo valor, uma vez que nós não queremos que sejam colocados na mesma posição.[1]

LINEAR SEARCH

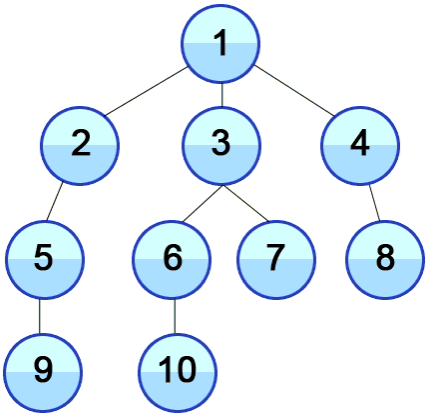
A busca linear como o próprio nome por si só indica, é a busca onde saímos buscando por todo conjunto de dados individualmente sendo uma notação O(N).

BINARY SEARCH

A busca binária tem uma representação idêntica vista na árvore binária onde para cada comparação o número de elementos a serem buscados divide por dois, onde a notação é equivalente a O(n log n)

BREADTH FIRST SEARCH

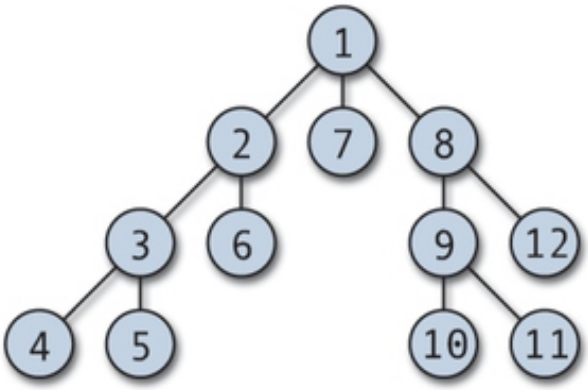
Este é um algoritimo de busca em grafos utilizado para realizar buscas em uma estrutura do tipo árvore de modo que é realizada uma varredura a parti dos nós que estão mais próximos do nó raiz.



Este tipo de busca é indicado quando os nós estão mais próximos do nó raiz do quê da base da árvore.

DEPTH FIRST SEARCH

Este algorítimo de busca, foca em explorar primeiramente nós a esquerda até que não exista mais nenhum além, em seguida ele percorre o trajeto de volta verificando os nós a direita também de forma recursiva.



Este tipo de busca é indicado quando os nós estão mais próximo da base da árvore, visto que o tempo de operação até que algum nó mais longe da raiz seja acessado é relativamente menor comparado ao algoritimo DFS.