**ESTRUTURA DE DADOS**

**Queue (Fila)**:

São estruturas lineares onde os dados são inseridos ao final e removidos do início, por exemplo, uma fila que suporte 5 elementos/nós, representada:

vamos marcar o primeiro da fila com verde, e o último com vermelho

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |  |

adicionamos um valor novo:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |

Ao fazer algum processamento ou realizar alguma ação, remove-se o primeiro da fila:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 |

Note que o primeiro espaço ainda existe, porém está vazio. O computador na verdade marca quem é o primeiro e quem é o último da fila, isso é menos custoso em termos de performance do que mover toda a lista para esquerda.

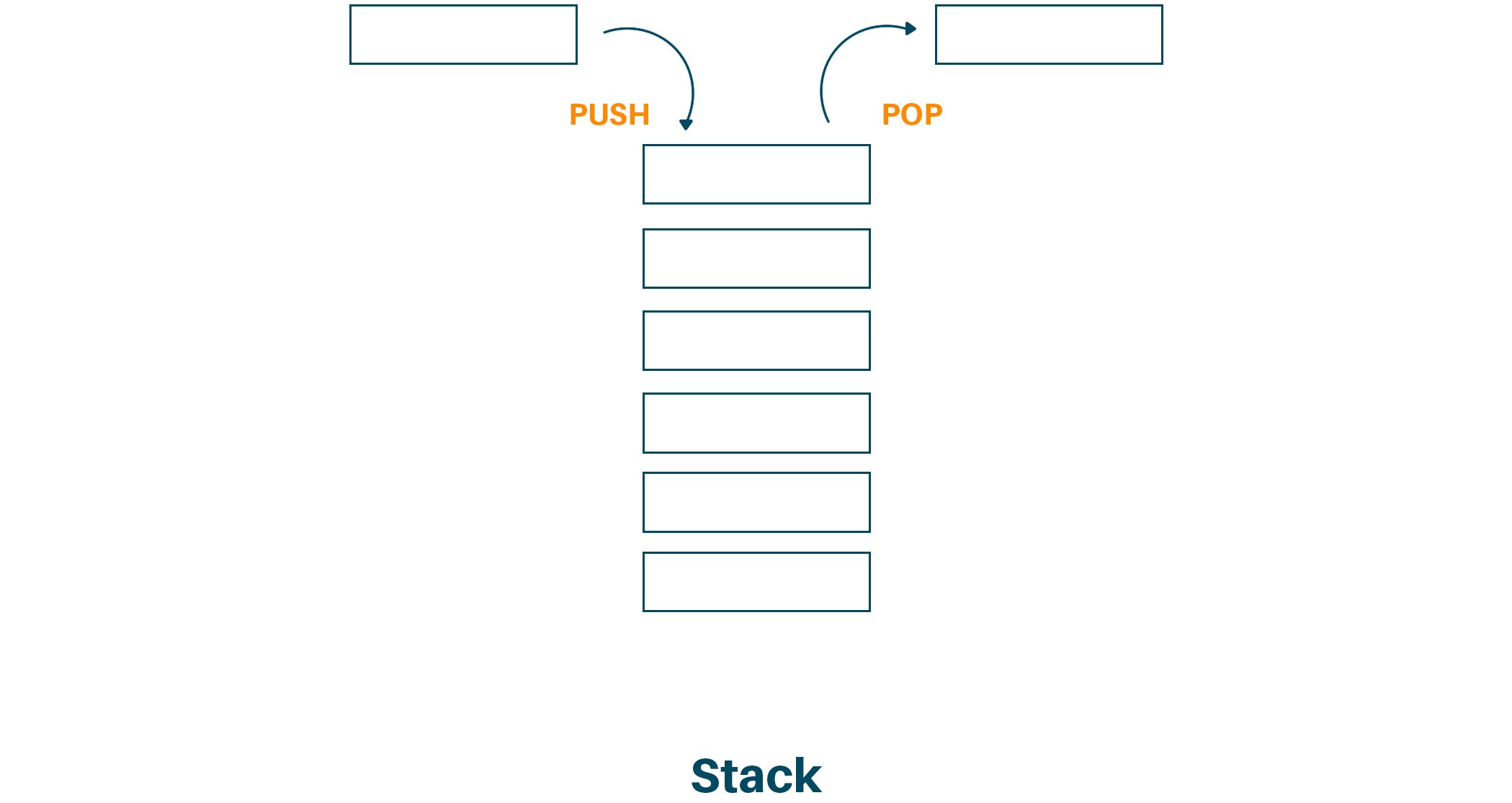
Portanto, se um novo valor for inserido agora, ele ocupará a primeira posição, porém será considerado o último da fila:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 6 | 2 | 3 | 4 | 5 |

**Stacks (pilhas)**

Pilhas de elementos são uma estrutura onde novos elementos são sempre adicionados ao topo de uma lista de elementos e o tamanho vai aumentando.

Um exemplo de pilha é o callstack da thread do Javascript, onde as execuções de código vão sendo “empilhadas”, aumentando o tamanho do callstack e sendo removidas do topo da lista a medida que vão sendo executadas. Utiliza princípio Last in First Out, ou seja o último a entrar é o primeiro a sair.



**Sets**

Sets são um tipo de estrutura de dados que armazena elementos únicos, ou seja, um set nunca terá dois elementos iguais em sua lista de elementos.

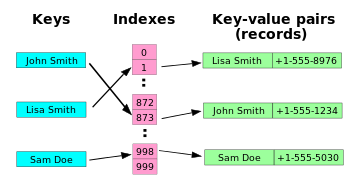
**Binary Search Tree (Árvore de busca binária)**

Uma estrutura de dados que se baseia em nodes (nós). Onde temos um nó raiz, que por ser uma árvore binária pode ter até dois nós filhos, que por suas vezes podem ter outros 2 nós filhos, e assim por diante. Para acessar um nó filho, é sempre preciso “navegar” a partir dos nós pais.  
Outro detalhe importante em uma Binary Search Tree, é que, a partir da raiz, os nós que seguem para esquerda, precisam ser sempre com valor inferior ao nó pai, e a direita, sempre maior que o nó pai. Com essa estrutura, uma busca pelos dados precisa metade das operações necessárias para encontrar um valor, desde que saibamos que o valor a ser encontrado é menor ou maior que o nó atual.



**Hash Tables**

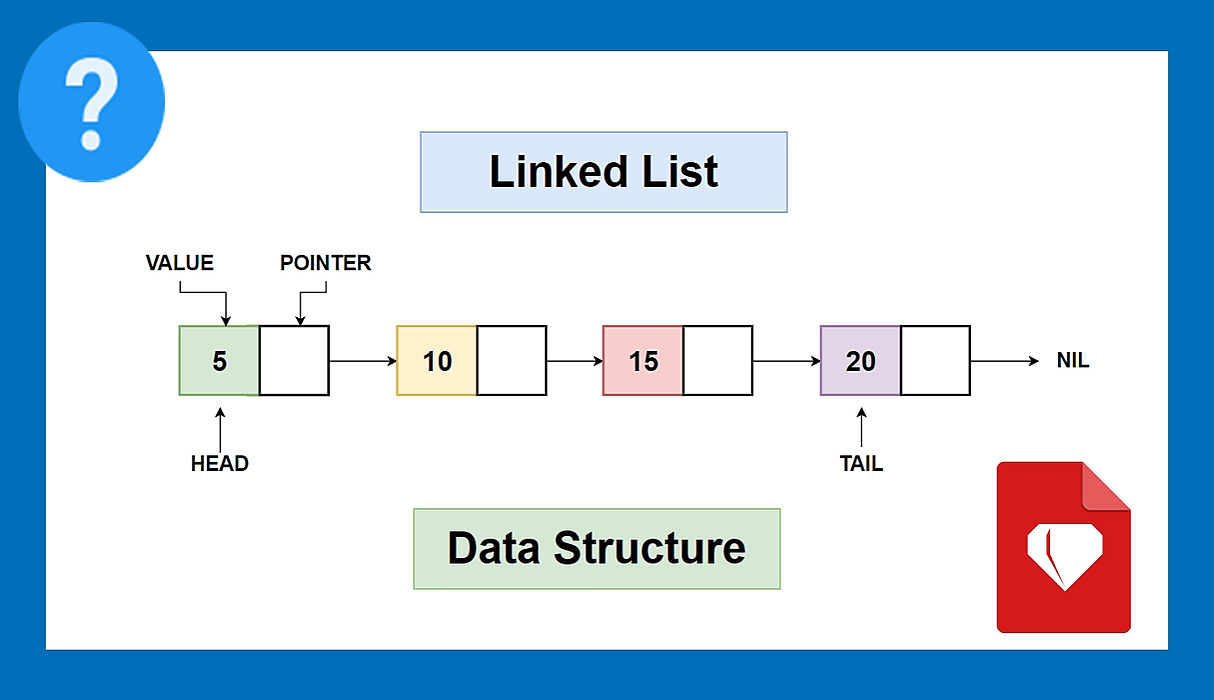
São usados para criação de arrays de associação ou mapeamento de chave e valor. São muito eficientes ,em termos de complexidade, para inserir, procurar e deletar elementos, tendo sua complexidade média na escala Big-O de O(1), o que significa um tempo sempre constante de processamento.  
 O que faz uma hash table é mapear as chaves em uma função hash, fazendo chaves strings serem mapeadas para índices. Assim, cada key será mapeada para um índice de memória específico, onde irá então armazenar o valor.



*Hash Table*: [[Lisa Smith, +1-555-8976], [John Smith, +1-555-1234], [Sam Doe, +1-555-5030]]

**Linked List**

É uma estrutura onde os dados são armazenados em nodes (nós). Cada nó contem duas informações, o dado em si e uma referência para o próximo nó, por isso é chamada de “Linked”.  
Suas vantagens são ter tamanho dinâmico e eficiência em inserção e deleção de dados. Porém peca na facilidade de acesso aos dados,, como em um array é possível acessar um índice diretamente, em uma Linked List não, sendo preciso correr cada nó até o nó desejado. Os elementos armazenados não estão na memória de forma linear, como em um array.



**Heap**

É uma espécie de árvore binária, porém nela os nodes pais tem relações específicas com os nodes filhos. Por exemplo, em uma Min Heap, os nodes pais sempre terão valores menores ou iguais aos nodes filhos, enquanto numa Max Heap, todos os nodes pais sempre com valores maiores ou iguais que os nodes filhos.



Heaps costumam serem implementadas como arrays, porém sem utilizar o index 0.  
  
Usando o exemplo da imagem acima, a max heap, ficaria representada em um array como:

*[<empty> , 100, 40, 50, 10, 15, 50, 40 ]*

Desta forma, podemos encontrar nós filhos com algumas equações simples:

O nó filho que esta na esquerda será: IndexPai \* 2  
 O nó filho que está na direita será: IndexPai \* 2 + 1  
E para encontrar o node pai de um node, usamos: indexNode / 2  
  
No exemplo acima , partindo do primeiro node, com valor 100, que está no index 1.  
seu filho a esquerda será: 1 \* 2 = 2 (index 2 do array, ou seja, valor 40)  
seu filho a direita será: 1 \* 2 + 1 = 3 (index 3 do array, ou seja, valor 50)  
se quero saber qual o node pai index 4 (valor 10), faço 4 / 2 = 2 (index 2, ou seja, valor 40)  
nesta divisão, em caso de valor quebrado, arrendondamos para baixo: 3 / 1 = 1.5 = 1.

Heaps são muito eficientes quando se trata de sorting (ordenação), pois o heap recria um array com todos elementos do array original, apenas removendo o menor elemento e colocando (push) no novo array, tornando ele reordenado.