

Anotações da Aula de Estrutura de Dados



Nesta que você acabou de assistir, foi abordado...

- Redimensionamento de matrizes
- Estrutura de Dados
- Listas Encadeadas
- Implementação de arrays
- Implementação de Listas Encadeadas
- Árvores
- Mais Estruturas de Dados

Quero compartilhar meu aprendizado e/ou minha dúvida...

















Ir para o Discord

Recomendamos que você leia as anotações da aula, isso pode te ajudar!

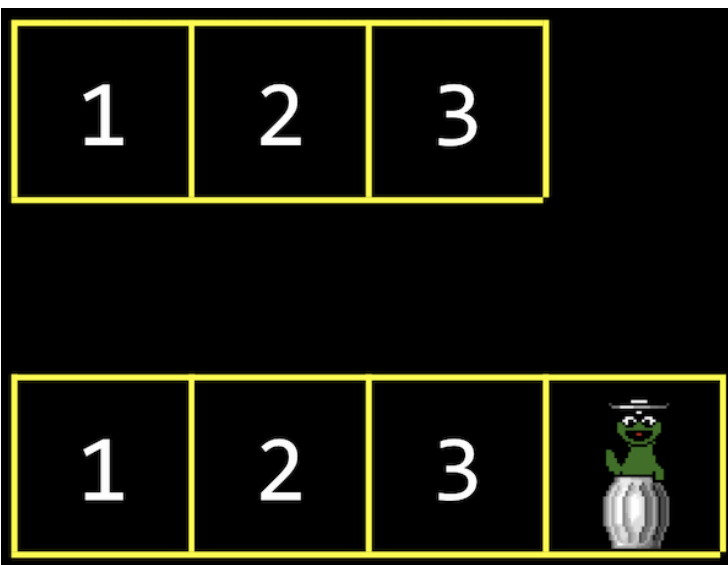
Redimensionamento de matrizes

Da última vez, aprendemos sobre ponteiros, **malloc** e outras ferramentas úteis para trabalhar com memória.

Na semana 2, aprendemos sobre matrizes, onde poderíamos armazenar o mesmo tipo de valor em uma lista, consecutivamente na memória. Quando precisamos inserir um elemento, precisamos aumentar o tamanho do array também. Mas, o espaço da memória ao lado desse array em nosso computador já pode estar em uso para alguns outros dados, como uma string:

							
	1	2	3	h	e	l	l
o	,		w	o	r	l	d
\0							

Uma solução pode ser alocar mais memória onde houver espaço suficiente e mover nosso array para lá. Mas precisaremos copiar nosso array lá, o que se torna uma operação com tempo de execução de $O(n)$, já que precisamos copiar cada um dos n elementos originais primeiro:



- O limite inferior da inserção de um elemento em um array seria $O(1)$, pois já podemos ter espaço para ele no array.

Estruturas de dados

As **estruturas de dados** são formas mais complexas de organizar os dados na memória, permitindo-nos armazenar informações em diferentes layouts.

Para construir uma estrutura de dados, vamos precisar de algumas ferramentas:

- **struct** para criar tipos de dados personalizados
- **.** para acessar propriedades em uma estrutura
- ***** para ir para um endereço na memória apontado por um ponteiro
- **->** para acessar propriedades em uma estrutura apontada por um ponteiro

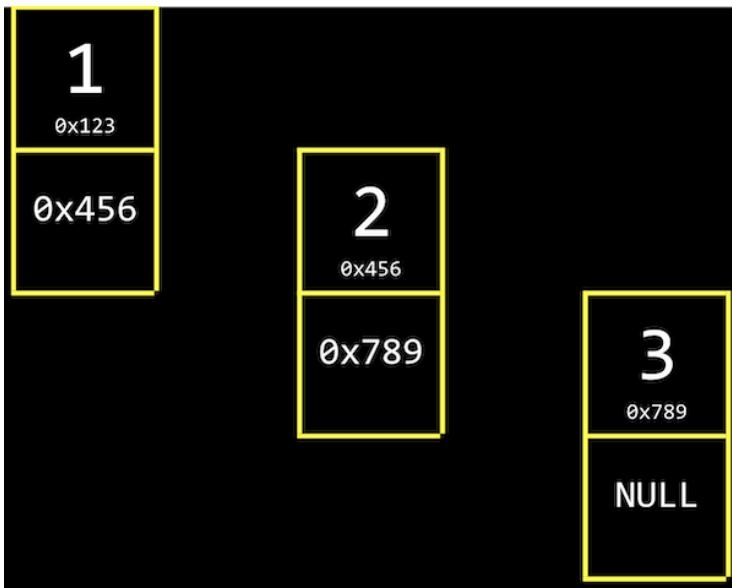
Linked lists

Com uma linked list (ou lista encadeada) , podemos armazenar uma lista de valores que pode ser facilmente aumentada, armazenando valores em diferentes partes da memória:

1 0x123				
		2 0x456		
				3 0x789

- Temos os valores 1, 2 e 3 , cada um em algum endereço na memória como 0x123, 0x456 e 0x789.
- Isso é diferente de uma matriz, pois nossos valores não estão mais próximos um do outro na memória. Podemos usar quaisquer locais da memória que estejam livres.

Para rastrear todos esses valores, precisamos vincular nossa lista, alocando, para cada elemento, memória suficiente para o valor que queremos armazenar e o endereço do próximo elemento:



- Próximo ao nosso valor de 1, por exemplo, também armazenamos um ponteiro, **0x456**, para o próximo valor. Chamaremos isso de **node** (ou **nó**), um componente de nossa estrutura de dados que armazena um valor e um ponteiro. Em C, implementaremos nossos nós com um struct.
- Para nosso último nó com valor 3, temos o ponteiro nulo, já que não há próximo elemento. Quando precisamos inserir outro nó, podemos apenas alterar aquele único ponteiro nulo para apontar para nosso novo valor.

Temos a desvantagem de precisar alocar o dobro de memória para cada elemento, a fim de gastar menos tempo adicionando valores. E não podemos mais usar a pesquisa binária, uma vez que nossos nós podem estar em qualquer lugar da memória. Só podemos acessá-los seguindo os ponteiros, um de cada vez.

No código, podemos criar nossa própria estrutura chamada **node** e precisamos armazenar tanto nosso valor, um **int** chamado **number**, quanto um ponteiro para o próximo **node**, chamado **next**:

```
typedef struct node
{
    int number;
    struct node *next;
}
node;
```

Iniciamos este com **typedef struct node** para que possamos nos referir a um **node** dentro de nosso struct.

Podemos construir uma lista vinculada no código começando com nosso struct. Primeiro, queremos lembrar de uma lista vazia, para que possamos usar o ponteiro nulo: **node *list = NULL;**

Para adicionar um elemento, primeiro precisamos alocar um pouco de memória para um nó e definir seus valores:

```
// Usamos sizeof (node) para obter a quantidade certa de memória para alocar, e
// malloc retorna um ponteiro que salvamos como
node n = malloc(sizeof(node));

// Queremos ter certeza de que malloc conseguiu obter memória para nós
if(n != NULL)
{
```

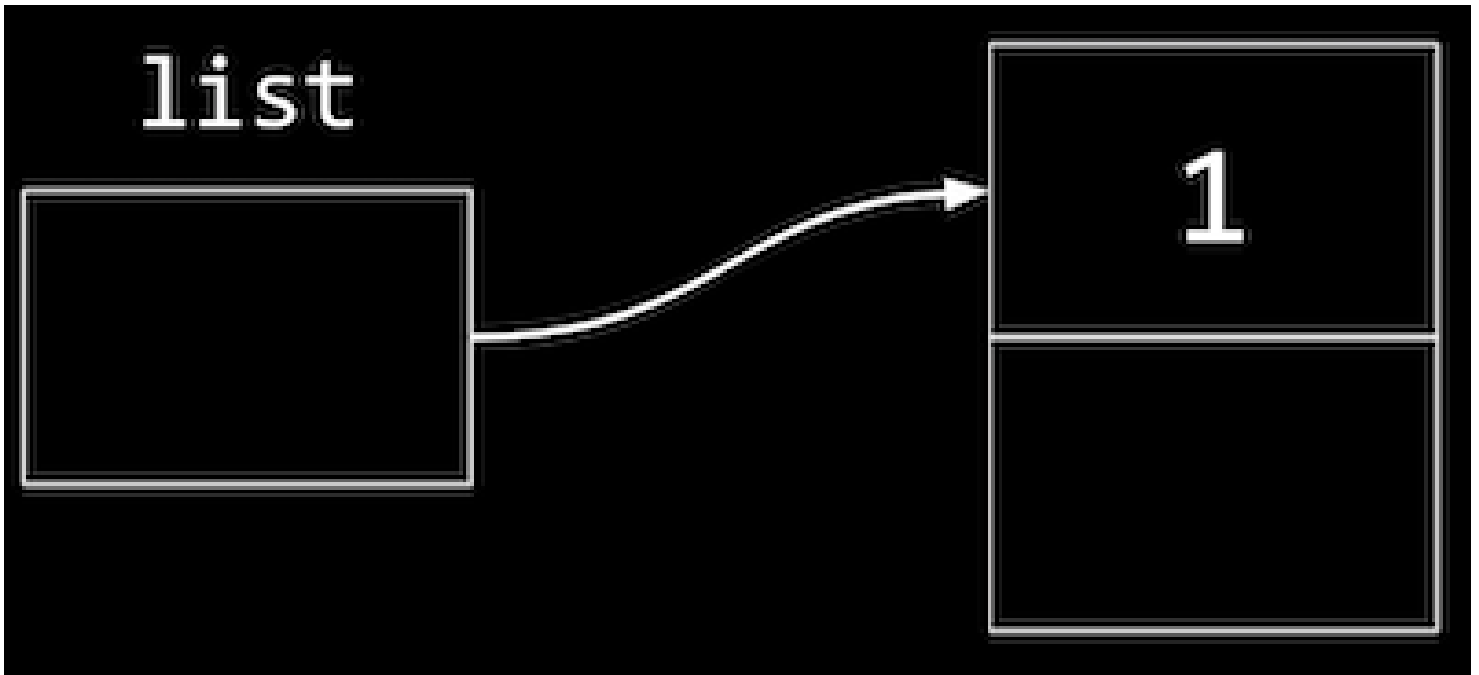
```

// Isso é equivalente a (n).number, onde primeiro vamos para o nó apontado
// para por n e, em seguida, defina a propriedade number. Em C, também podemos usar este
// notação de seta
n->number = 1;

// Então precisamos ter certeza de que o ponteiro para o próximo nó em nossa lista
// não é um valor lixo, mas o novo nó não apontará para nada (por enquanto)
n->next = NULL;
}

```

Agora nossa lista precisa apontar para este nó: `list = n;`



Para adicionar à lista, criaremos um novo nó da mesma maneira, alocando mais memória:

```

n = malloc(sizeof(node));

if(n != NULL)
{
    n->number = 2;
    n->next = NULL;
}

```

Mas agora precisamos atualizar o ponteiro em nosso primeiro nó para apontar para nosso novo `n`:

```
list->next = n;
```

Para adicionar um terceiro nó, nós vamos fazer o mesmo, seguindo o ponteiro `next` na nossa lista em primeiro lugar, em seguida, definir o ponteiro `next` para lá para apontar para o novo nó:

```

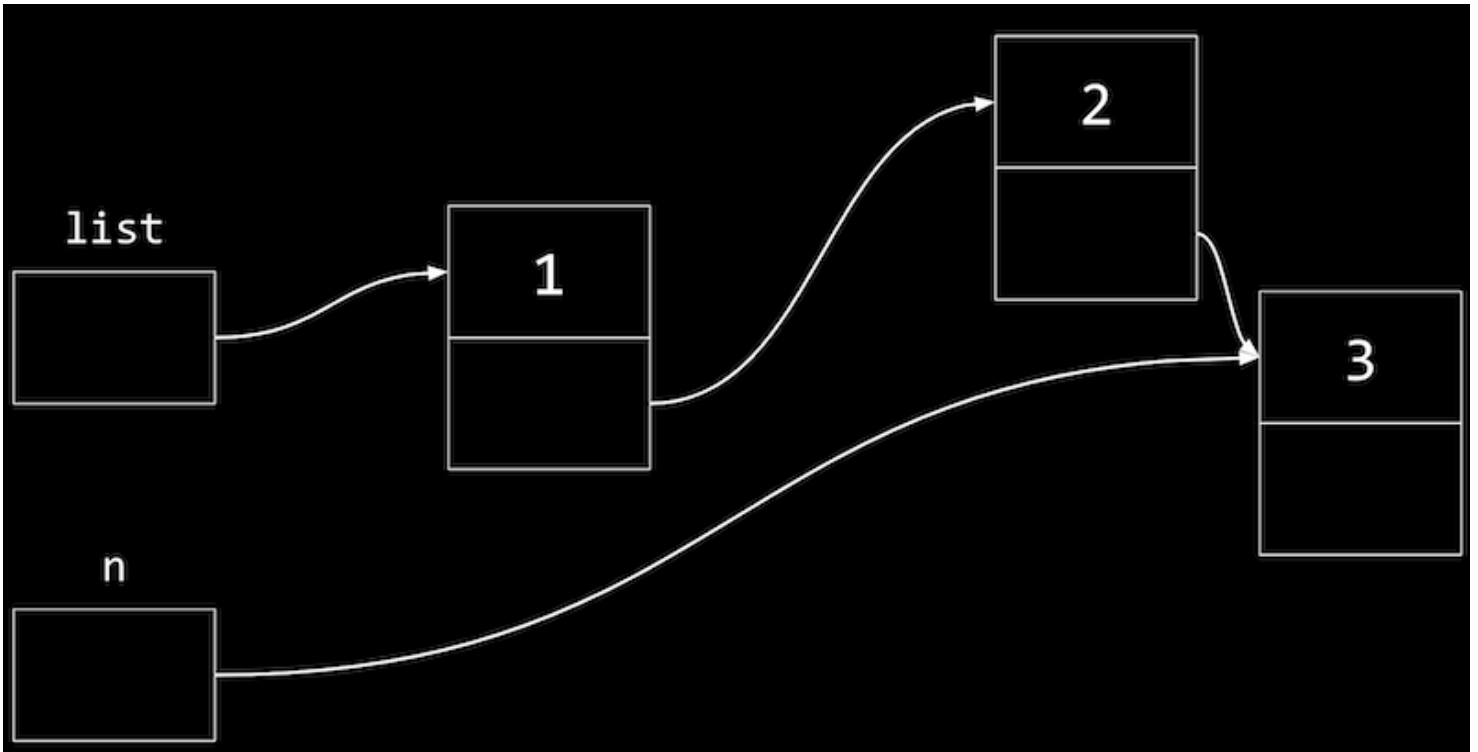
n = malloc(sizeof(node));

if(n != NULL)
{
    n->number = 3;
    n->next = NULL;
}

```

```
}  
list->next->next = n;
```

Graficamente, nossos nós na memória se parecem com isto:



- **n** é uma variável temporária, apontando para nosso novo nó com valor 3.
- Queremos que o ponteiro em nosso nó com valor 2 aponte para o novo nó também, então começamos da **list**(que aponta para o nó com valor 1), seguimos o ponteiro **next** para chegar ao nosso nó com valor 2 e atualizamos o ponteiro **next** para apontar para **n**.

Como resultado, pesquisar uma lista encadeada também terá um tempo de execução de $O(n)$, uma vez que precisamos olhar todos os elementos em ordem seguindo cada ponteiro, mesmo se a lista estiver classificada. A inserção em uma lista encadeada pode ter um tempo de execução de $O(1)$, se inserirmos novos nós no início da lista.

Implementando Vetores

Vamos ver como podemos implementar o redimensionamento de uma matriz:

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

int main(void)
{
    // Use malloc para alocar espaço suficiente para uma matriz com 3 inteiros
    int list = malloc(3 * sizeof(int));
    if (list == NULL)
    {
        return 1;
    }
}
```

```

// Defina os valores em nosso array
list[0] = 1;
list[1] = 2;
list[2] = 3;

// Agora, se quisermos armazenar outro valor, podemos alocar mais memória
int tmp = malloc(4 * sizeof(int));
if (tmp == NULL)
{
    free(list);
    return 1;
}

// Copie a lista de tamanho 3 para a lista de tamanho 4
for(int i = 0; i < 3; i++)
{
    tmp[i] = list[i];
}

// Adicionar novo número à lista de tamanho 4
tmp[3] = 4;

// Lista original grátis de tamanho 3
free(list);

// Lembre-se da nova lista de tamanho 4
list = tmp;

// Imprimir lista
for (int i = 0; i < 4; i++)
{
    printf("%i\n", list[i]);
}

// Free nova lista
free(list);
}

```

Lembre-se de que o **malloc** aloca e libera memória da área de heap. Acontece que podemos chamar outra função de biblioteca, **realloc**, para realocar alguma memória que alocamos anteriormente:

```
int tmp = realloc (list, 4 * sizeof ( int ));
```

- E **realloc** copia nosso antigo array, **list**, para nós em um pedaço maior de memória do tamanho que passamos. Se acontecer de haver espaço depois de nosso pedaço de memória existente, vamos obter o mesmo endereço de volta, mas com a memória depois de alocado à nossa variável também.

Implementando Listas Encadeadas

Vamos combinar nossos trechos de código anteriores em um programa que implementa uma lista vinculada:

```

#include <stdio.h>
#include <stdlib.h>

// Representando um nó
typedef struct node
{
    int number; struct node next;
}
node;

```

```

int main(void)
{
    // Lista de tamanho 0. Inicializamos o valor para NULL explicitamente, então há
    // nenhum valor de lixo para nossa variável de lista
    node list = NULL;

    // Alocar memória para um node, n
    node n = malloc(sizeof(node));
    if(n == NULL)
    {
        return 1;
    }

    // Definir o valor e ponteiro no nosso node
    n->number = 1;
    n->next = NULL;

    // Adicione o nó n apontando lista para ele, uma vez que só temos um nó até agora
    list = n;

    // Aloque memória para outro nó, e podemos reutilizar nossa variável n para
    // aponte para ele, uma vez que a lista já aponta para o primeiro nó
    n = malloc(sizeof(node));
    if(n == NULL)
    {
        free(list);
        return 1;
    }

    // Defina os valores em nosso novo nó
    n->number = 2;
    n->next = NULL;

    // Atualize o ponteiro em nosso primeiro nó para apontar para o segundo nó
    list->next = n;

    // Alocar memória para um terceiro nó
    n = malloc(sizeof(node));

    if(n == NULL){
        // Libere nossos outros nós
        free(list->next);
        free(list);
        return 1;
    }
    n->number = 3;
    n->next = NULL;

    // Siga o próximo ponteiro da lista para o segundo nó e atualize
    // o próximo ponteiro para apontar para n
    list->next->next = n;

    // Imprime a lista usando um loop, usando uma variável temporária, tmp, para apontar
    // para listar, o primeiro nó. Então, toda vez que examinamos o loop, usamos
    // tmp = tmp-> next para atualizar nosso ponteiro temporário para o próximo nó.
    // continue enquanto tmp aponta para algum lugar, parando quando chegarmos a
    // o último nó e tmp-> next é nulo.
    for(node tmp = list; tmp != NULL; tmp = tmp->next)
    {
        printf("%i\n", tmp->number);
    }

    // Libere a lista, usando um loop while e uma variável temporária para apontar
    // para o próximo nó antes de liberar o atua
    while(list != NULL)
    {
        // Nós apontamos para o próximo nó primeiro
        node *tmp = list->next;

        // Então, podemos liberar o primeiro nó free(list);

```



```

// Agora podemos definir a lista para apontar para o próximo nó
list = tmp;

// Se a lista for nula, quando não houver mais nós restantes, nosso loop while irá parar
}
}

```

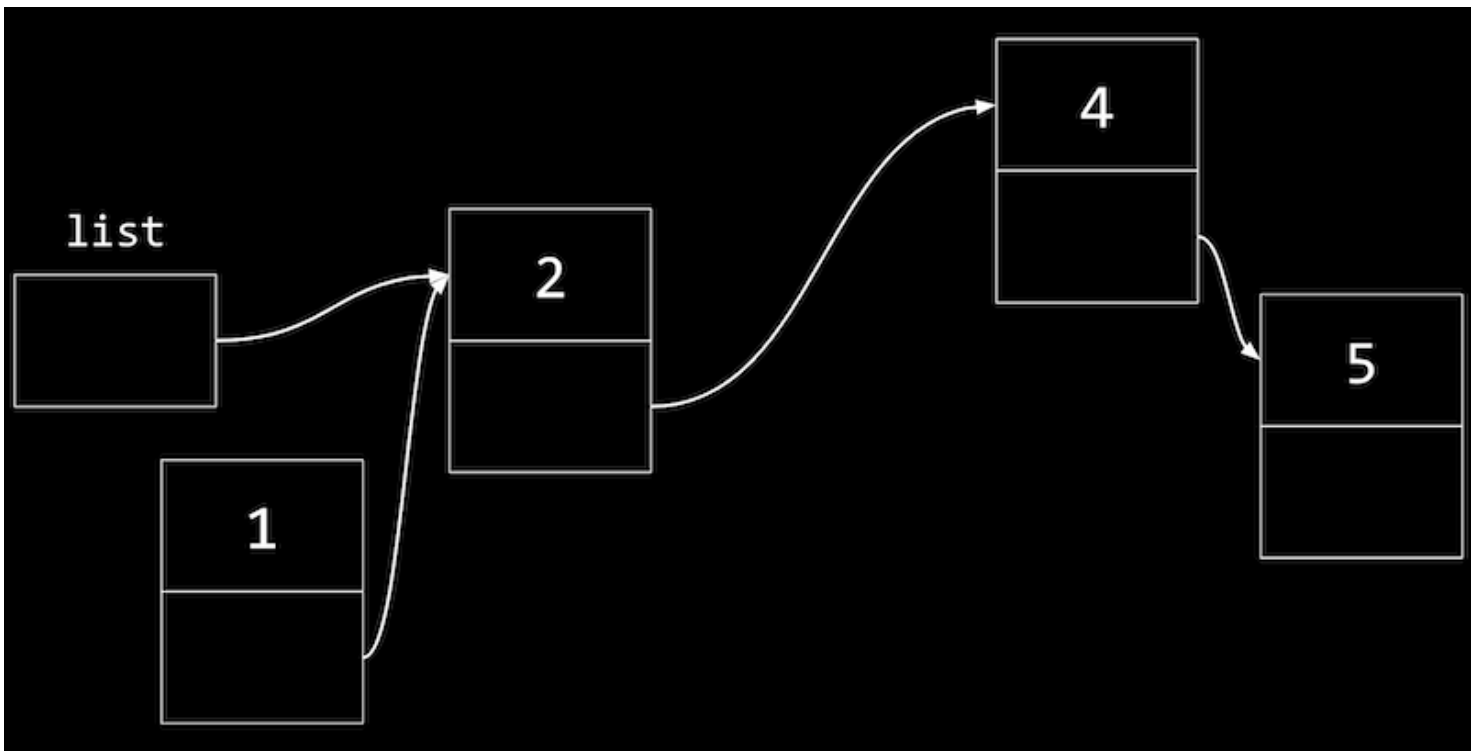
Se quisermos inserir um nó à frente de nossa lista vinculada, precisaremos atualizar cuidadosamente nosso nó para apontar para o seguinte, antes de atualizar a variável de lista. Caso contrário, perderemos o resto da nossa lista:

```

// Aqui, estamos inserindo um nó na frente da lista, então queremos que seu
// próximo ponteiro para apontar para a lista original. Então podemos mudar a lista para
// aponta para n.
n->próximo = lista; lista = n;

```

A princípio, teremos um nó com valor 1 apontando para o início de nossa lista, um nó com valor 2



- Agora podemos atualizar nossa variável **list** para apontar para o nó com valor **1**, e não perder o resto de nossa lista.

Da mesma forma, para inserir um nó no meio de nossa lista, mudamos o ponteiro **next** do novo nó primeiro para apontar para o resto da lista e, em seguida, atualizamos o nó anterior para apontar para o novo nó.

Uma lista vinculada demonstra como podemos usar ponteiros para construir estruturas de dados flexíveis na memória, embora estejamos apenas visualizando em uma dimensão.

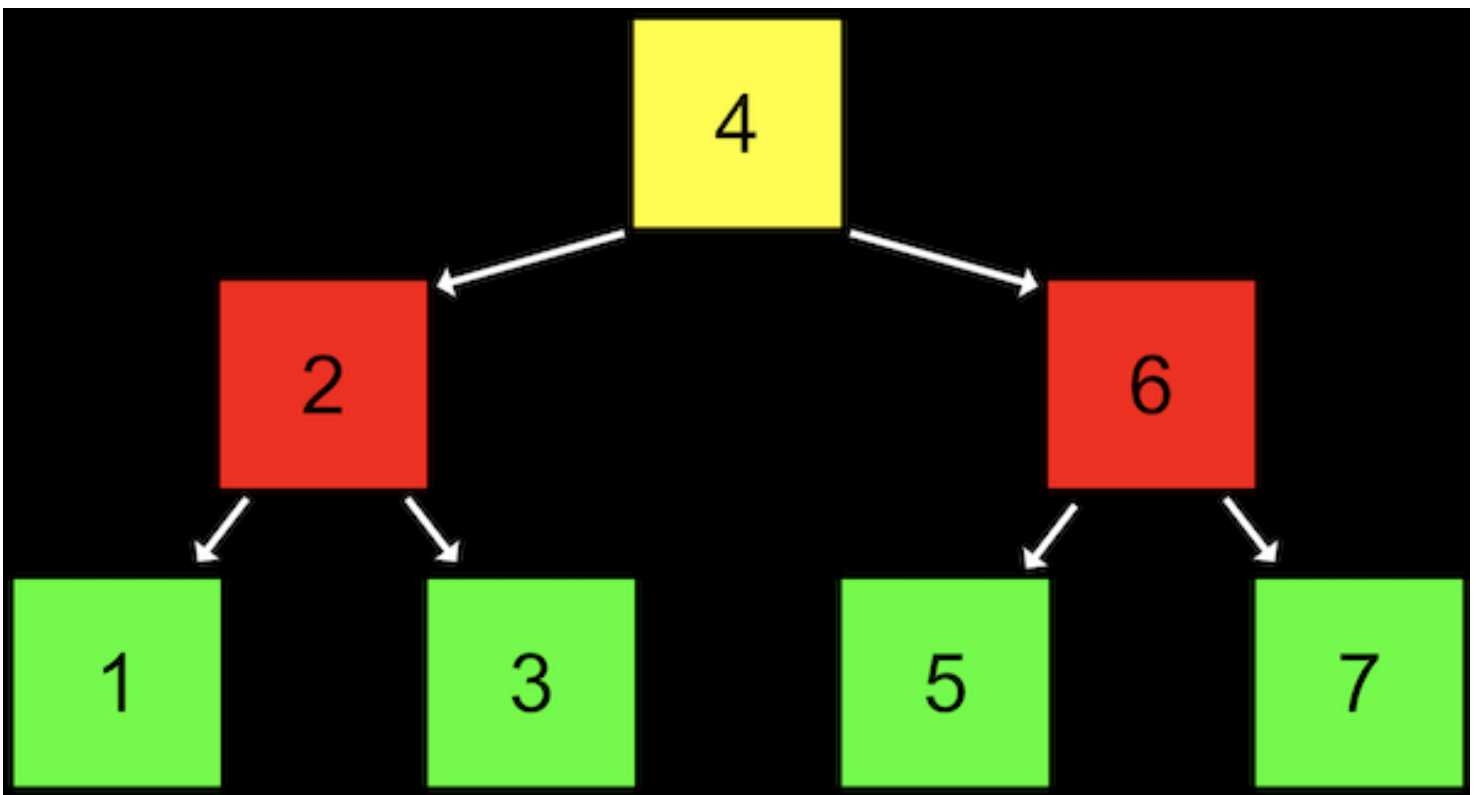
Árvores

Com um array ordenado, podemos usar a pesquisa binária para encontrar um elemento, começando no meio (amarelo), depois no meio da metade (vermelho) e, finalmente, à esquerda ou à direita (verde), conforme necessário:



- Com um array, podemos acessar elementos aleatoriamente no tempo $O(1)$, uma vez que podemos usar a aritmética para ir para um elemento em qualquer índice.

Uma **árvore(tree)** é outra estrutura de dados onde cada nó aponta para dois outros nós, um à esquerda (com um valor menor) e um à direita (com um valor maior):



- Observe que agora visualizamos essa estrutura de dados em duas dimensões (mesmo que os nós na memória possam estar em qualquer local).
- E podemos implementar isso com uma versão mais complexa de um nó em uma lista vinculada, onde cada nó tem não um, mas dois ponteiros para outros nós. Todos os valores à esquerda de um nó são menores e todos os valores dos nós à direita são maiores, o que permite que isso seja usado como uma **árvore de busca binária**. E a própria estrutura de dados é definida recursivamente, então podemos usar funções recursivas para trabalhar com ela.
- Cada nó tem no máximo dois **filhos(children)**, ou nós para os quais está apontando.

- E como uma lista vinculada, queremos manter um ponteiro apenas para o início da lista, mas neste caso queremos apontar para a **raíz(root)**, ou nó central superior da árvore (o 4).

Podemos definir um nó não com um, mas com dois ponteiros:

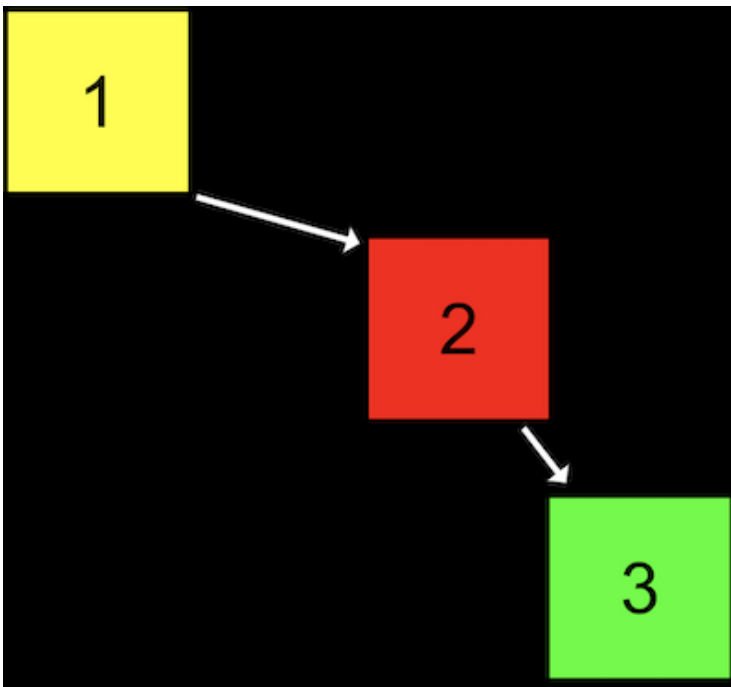
```
typedef struct node
{
    int number;
    struct node left;
    struct node right;
}
node;
```

E escreva uma função para pesquisar recursivamente em uma árvore:

```
// árvore é um ponteiro para um nó que é a raiz da árvore que estamos pesquisando.
// número é o valor que estamos tentando encontrar na árvore.
bool search(node *tree, int number)
{
    // Primeiro, nos certificamos de que a árvore não é NULL, se alcançamos um nó
    // na parte inferior, ou se nossa árvore estiver totalmente vazia
    if(tree == NULL){
        return false;
    }
    // Se estivermos procurando por um número menor que o número da árvore,
    // pesquisa do lado esquerdo, usando o nó à esquerda como a nova raiz
    else if(number < tree->number)
    {
        return search(tree->left, number);
    }
    // Caso contrário, pesquise no lado direito, usando o nó à direita como a nova raiz
    else if (number > tree->number)
    {
        return search(tree->right, number);
    }
    // Finalmente, encontramos o número que procuramos, portanto, podemos retornar true
    // Podemos simplificar isso para apenas "outro", uma vez que não há outro caso possível
    else if(number == tree->number)
    {
        return true;
    }
}
```

Com uma árvore de pesquisa binária, incorremos no custo de ainda mais memória, já que cada nó agora precisa de espaço para um valor e dois ponteiros. A inserção de um novo valor levaria tempo $O(\log n)$, uma vez que precisamos encontrar os nós entre os quais ele deve ficar.

No entanto, se adicionarmos nós suficientes, nossa árvore de pesquisa pode começar a se parecer com uma lista vinculada:



- Começamos nossa árvore com um nó com valor 1, depois adicionamos o nó com valor 2 e, finalmente, adicionamos o nó com valor 3. Mesmo que essa árvore siga as restrições de uma árvore de pesquisa binária, não é tão eficiente quanto poderia ser.
- Podemos tornar a árvore balanceada, ou ótima, tornando o nó com valor 2 o novo nó raiz. Cursos mais avançados cobrirão estruturas de dados e algoritmos que nos ajudam a manter as árvores equilibradas conforme os nós são adicionados.

Mais estruturas de dados

Uma estrutura de dados com tempo quase constante, $O(1)$ search é uma **tabela hash**, que é essencialmente um *array* de listas encadeadas. Cada lista encadeada na matriz possui elementos de uma determinada categoria.

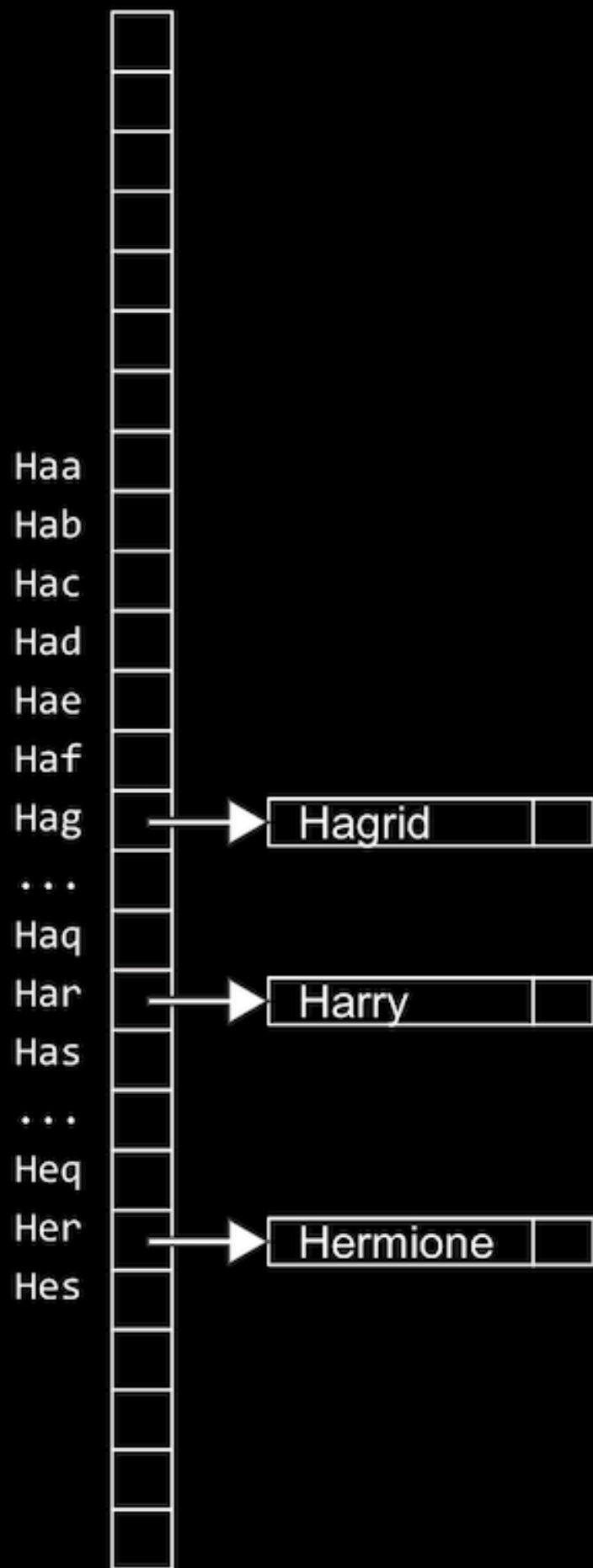
Por exemplo, podemos ter muitos nomes e podemos classificá-los em uma matriz com 26 posições, uma para cada letra do alfabeto:



- Como temos acesso aleatório com matrizes, podemos definir elementos e índices em um local, ou intervalo, na matriz rapidamente.
- Um local pode ter vários valores correspondentes, mas podemos adicionar um valor a outro valor, já que eles são nós em uma lista vinculada, como vemos com Hermione, Harry e Hagrid. Não precisamos aumentar o tamanho de nossa matriz ou mover qualquer um de nossos outros valores.

Isso é chamado de tabela hash porque usamos uma **função hash**, que pega alguma entrada e mapeia de forma determinística para o local em que deveria ir. Em nosso exemplo, a função hash apenas retorna um índice correspondente à primeira letra do nome, como como **0** para “Alvo” e **25** para “Zacarias”.

Mas, na pior das hipóteses, todos os nomes podem começar com a mesma letra, então podemos terminar com o equivalente a uma única lista vinculada novamente. Podemos olhar para as duas primeiras letras e alocar baldes suficientes para 26 *26 valores de hash possíveis, ou mesmo as três primeiras letras, exigindo 26 * 26 baldes:*



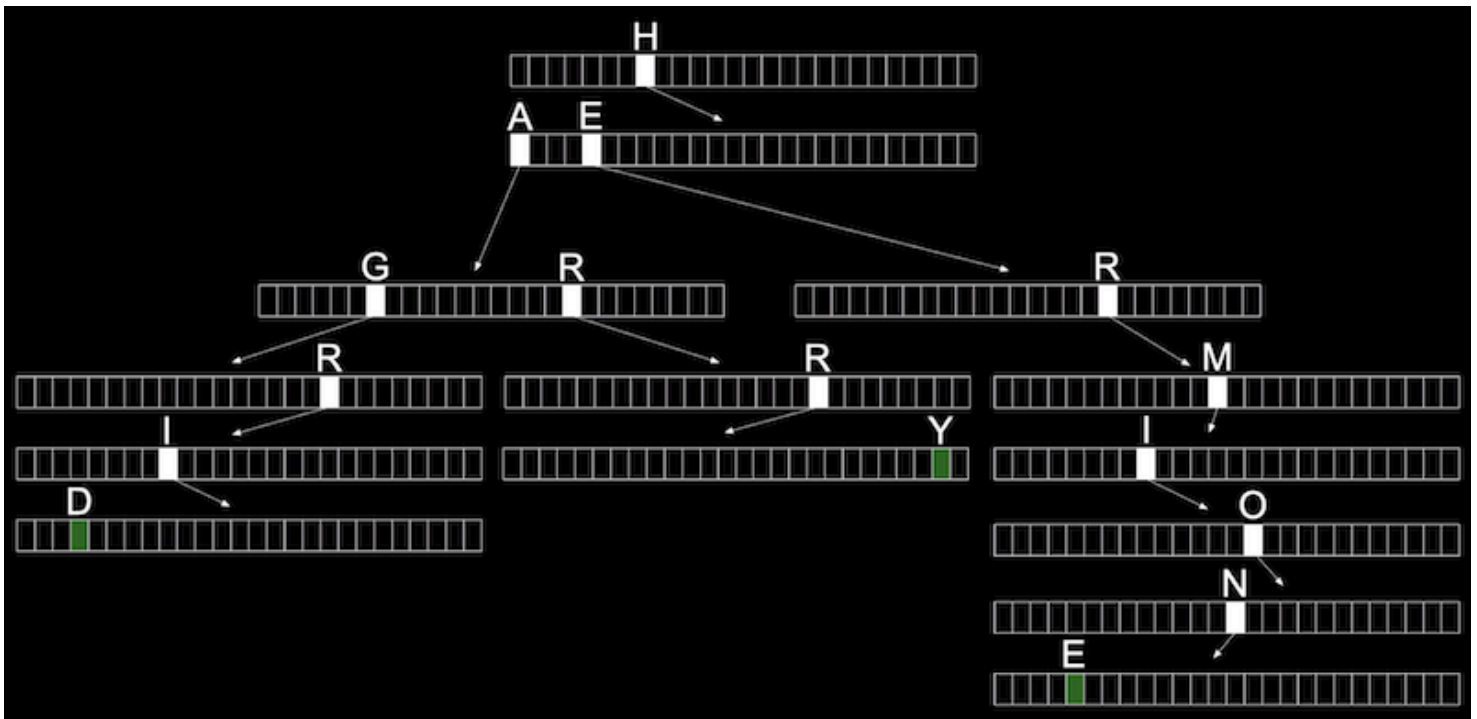
- Agora, estamos usando mais espaço na memória, já que alguns desses depósitos estarão vazios, mas é mais provável que necessitemos apenas uma etapa para procurar um valor, reduzindo nosso tempo de execução para pesquisa.

Para classificar algumas cartas de jogo padrão, também podemos começar colocando-as em pilhas por naipe, de espadas, ouros, copas e paus. Então, podemos classificar cada pilha um pouco mais rapidamente.

Acontece que o pior caso de tempo de execução para uma tabela hash é $O(n)$, uma vez que, à medida que n fica muito grande, cada depósito terá valores da ordem de n , mesmo que tenhamos centenas ou milhares de depósitos. Na prática, porém, nosso tempo de execução será mais rápido, pois estamos dividindo nossos valores em vários depósitos.

No conjunto de problemas 5, seremos desafiados a melhorar o tempo de execução do mundo real de pesquisa de valores em nossas estruturas de dados, ao mesmo tempo em que equilibramos nosso uso de memória.

Podemos usar outra estrutura de dados chamada **trie** (pronuncia-se "try" e é uma abreviação de "recuperação"). Um trie é uma árvore com matrizes como nós:



- Cada array terá cada letra, AZ, armazenada. Para cada palavra, a primeira letra apontará para um array, onde a próxima letra válida apontará para outro array, e assim por diante, até chegarmos a um valor booleano indicando o final de uma palavra válida, marcada em verde acima. Se nossa palavra não estiver no teste, então uma das matrizes não terá um ponteiro ou caractere de terminação para nossa palavra.
- No trie acima, temos as palavras Hagrid, Harry e Hermione.
- Agora, mesmo que nossa estrutura de dados tenha muitas palavras, o tempo máximo de pesquisa será apenas o comprimento da palavra que estamos procurando. Isso pode ser um máximo fixo, então podemos ter $O(1)$ para pesquisa e inserção.

- O custo disso, porém, é que precisamos de muita memória para armazenar ponteiros e valores booleanos como indicadores de palavras válidas, embora muitos deles não sejam usados.

Existem construções de nível ainda mais alto, **estruturas de dados abstratas**, onde usamos nossos blocos de construção de arrays, listas vinculadas, tabelas de hash e tentamos implementar uma solução para algum problema.

Por exemplo, uma estrutura de dados abstrata é uma **queue (ou fila)**, como uma fila de pessoas esperando, onde o primeiro valor que colocamos são os primeiros valores que são removidos, ou first-in-first-out (FIFO). Para adicionar um valor que **enfileira-lo**, e para remover um valor que **desenfileira-lo**. Essa estrutura de dados é abstrata porque é uma ideia que podemos implementar de diferentes maneiras: com um array que redimensionamos à medida que adicionamos e removemos itens, ou com uma lista vinculada onde acrescentamos valores ao final.

Uma estrutura de dados "oposta" seria um **stack (ou pilha)**, onde os itens adicionados mais recentemente são removidos primeiro: último a entrar , primeiro a sair (UEPS). Em uma loja de roupas, podemos pegar, ou **abrir** , o suéter de cima de uma pilha, e novos suéteres são adicionados, ou **empurrados**, para cima também.

Outro exemplo de estrutura de dados abstrata é um **dicionário** , onde podemos mapear chaves para valores, como palavras para suas definições. Podemos implementar um com uma tabela hash ou um array, levando em consideração a relação entre tempo e espaço.