Este é o CS50

Introdução à Ciência da Computação (CS50)

OpenCourseWare

Doar (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan)
(https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

Aula 5

Bem-vindo!

- Nas semanas anteriores, você aprendeu os fundamentos da programação.
- Tudo o que você aprendeu em C permitirá que você implemente esses blocos de construção em linguagens de programação de nível superior, como Python.

- A cada semana, os conceitos se tornam mais e mais desafiadores, como uma colina que se torna cada vez mais íngreme. Esta semana, o desafio se equilibra à medida que exploramos estruturas de dados.
- Até o momento, você aprendeu como um array pode organizar dados na memória.
- Hoje, vamos falar sobre como organizar dados na memória e sobre as possibilidades de design que surgem com o seu conhecimento crescente.

Estruturas de dados

- *Estruturas de dados* são essencialmente formas de organização na memória.
- Existem muitas maneiras de organizar dados na memória.
- *Tipos de dados abstratos* são aqueles que podemos imaginar conceitualmente. Ao aprender sobre ciência da computação, muitas vezes é útil começar com essas estruturas de dados conceituais. Aprendê-las facilitará posteriormente a compreensão de como implementar estruturas de dados mais concretas.

Filas

- *Filas* são uma forma de estrutura de dados abstrata.
- As filas têm propriedades específicas. Ou seja, elas seguem a *regra FIFO* , ou seja, "primeiro a entrar, primeiro a sair". Imagine-se numa fila para um brinquedo em um parque de diversões. A primeira pessoa da fila é a primeira a ir no brinquedo. A última pessoa é a última a ir no brinquedo.
- As filas possuem ações específicas associadas a elas. Por exemplo, um item pode ser *enfileirado* ; ou seja, o item pode entrar na fila. Além disso, um item pode ser *removido* da fila ou sair dela assim que chegar ao início da fila.
- Em termos de código, você pode imaginar uma fila da seguinte forma:

```
const int CAPACITY = 50;

typedef struct
{
    person people[CAPACITY];
    int size;
}
queue;
```

Observe que um array chamado `people` é do tipo `std::string` `person` . O `CAPACITY` valor `std::string` representa a altura máxima da pilha. O inteiro `std::string` `size` indica o quão cheia a fila está, independentemente de sua capacidade *máxima* .

Pilhas

- Filas são o oposto de *pilhas* . Fundamentalmente, as propriedades de uma pilha são diferentes das de uma fila. Especificamente, ela segue *o princípio LIFO* , ou "último a entrar, primeiro a sair". Assim como empilhar bandejas em um refeitório, a bandeja que é colocada por último em uma pilha é a primeira a ser retirada.
- Pilhas possuem ações específicas associadas a elas. Por exemplo, " *empurrar*" coloca algo no topo de uma pilha. *"Desempilhar* " remove algo do topo da pilha.
- Em código, você pode imaginar uma pilha da seguinte forma:

```
const int CAPACITY = 50;

typedef struct
{
    person people[CAPACITY];
    int size;
}
stack;
```

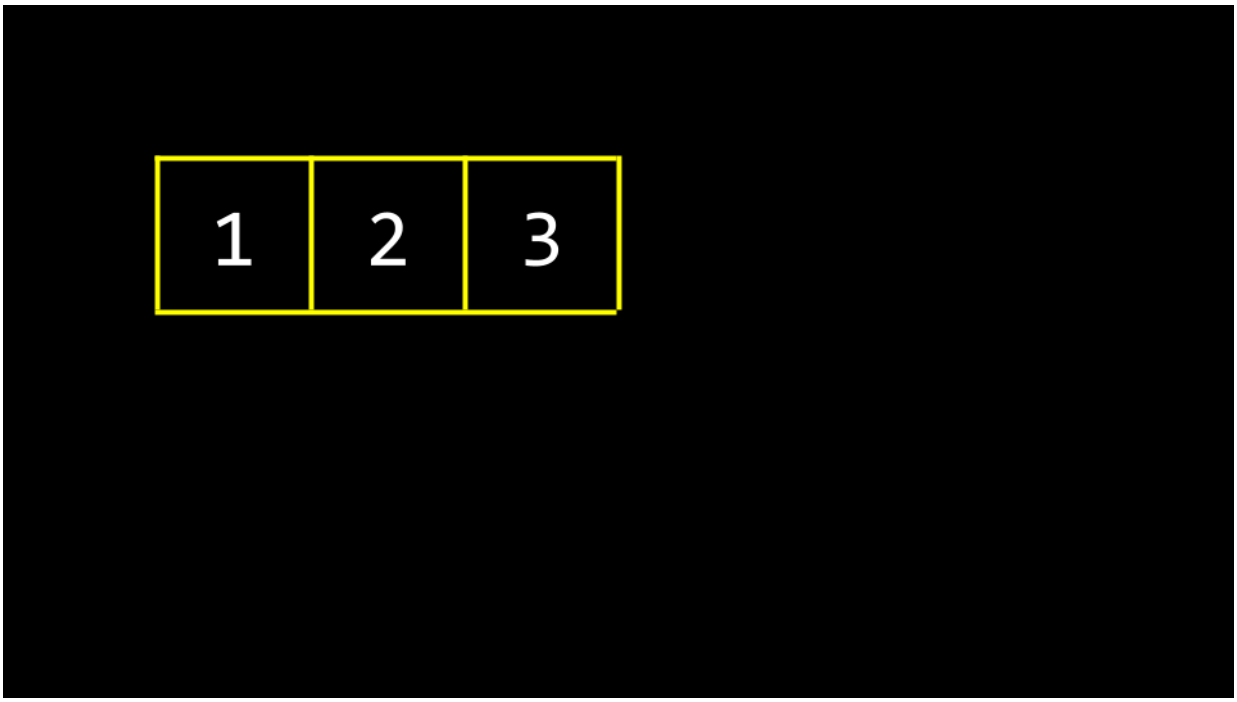
  Observe que um array chamado `people` é do tipo `std::array` `person` . O `CAPACITY` valor `std::array` representa a altura máxima que a pilha pode atingir. O inteiro `std::array` *representa* `size` o quão cheia a pilha está de fato, independentemente de sua capacidade máxima . Observe que este código é idêntico ao código da fila.
- Você pode imaginar que o código acima tem uma limitação. Como a capacidade do array é sempre predeterminada neste código, a pilha pode sempre ficar sobrecarregada. Você pode imaginar que apenas um dos 5000 espaços da pilha seja utilizado.
- Seria ótimo se nossa pilha fosse dinâmica – capaz de crescer à medida que itens fossem adicionados a ela.

Jack aprende os fatos

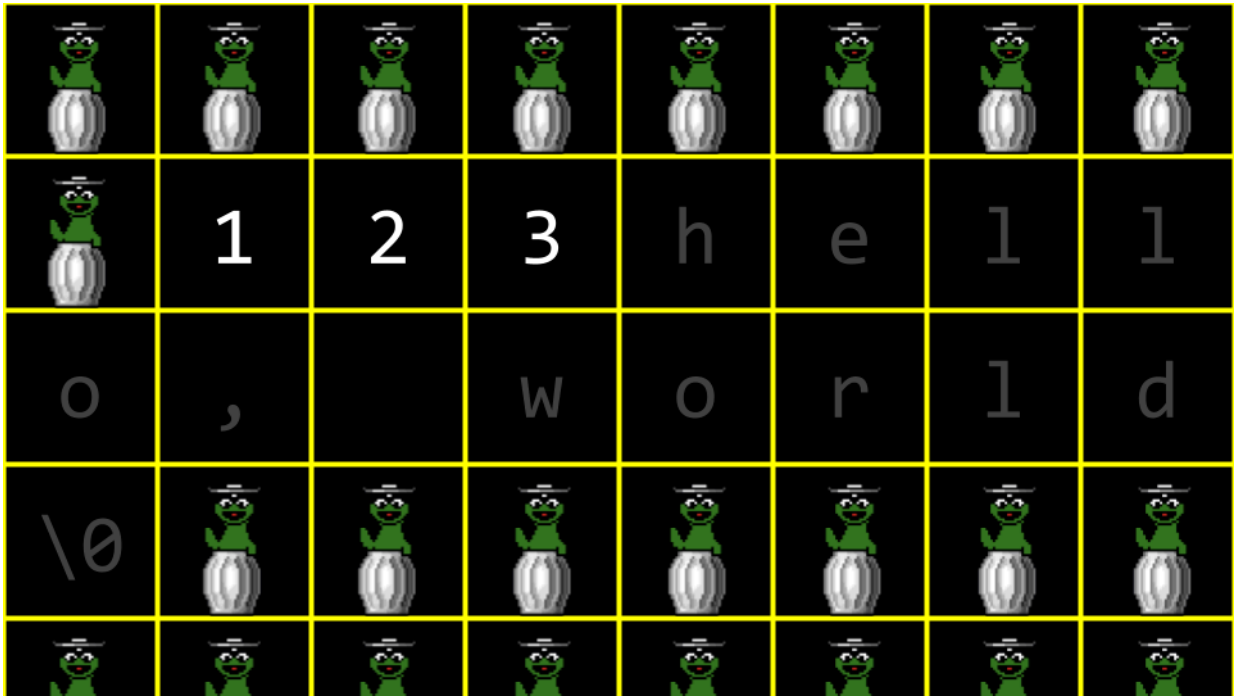
- Assistimos a um vídeo chamado "Jack aprende os fatos", (https://www.youtube.com/watch?v=ItAG3s6KIEI) da professora Shannon Duvall, da Universidade Elon.

Redimensionando matrizes

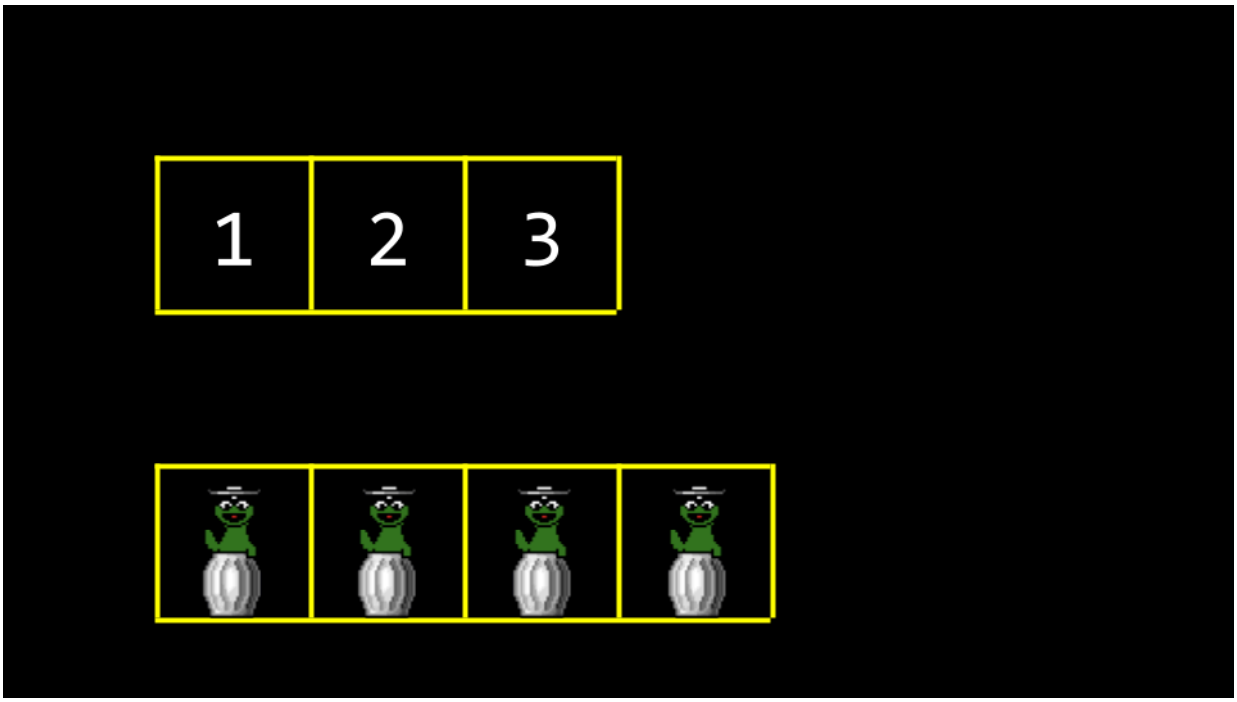
- Voltando à Semana 2, apresentamos a vocês a primeira estrutura de dados.
- Um array é um bloco de memória contígua.
- Você pode imaginar um array da seguinte forma:



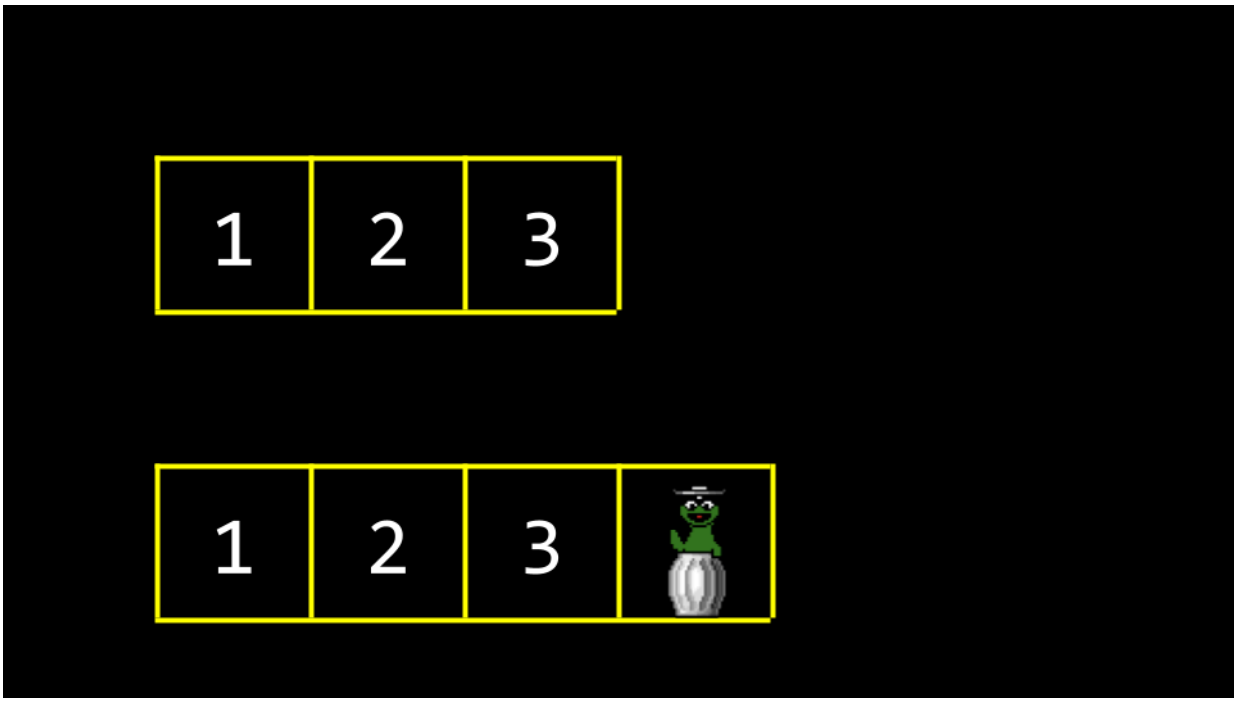
- Na memória, existem outros valores armazenados por outros programas, funções e variáveis. Muitos desses valores podem ser lixo, valores que foram utilizados em algum momento, mas que agora estão disponíveis para uso.



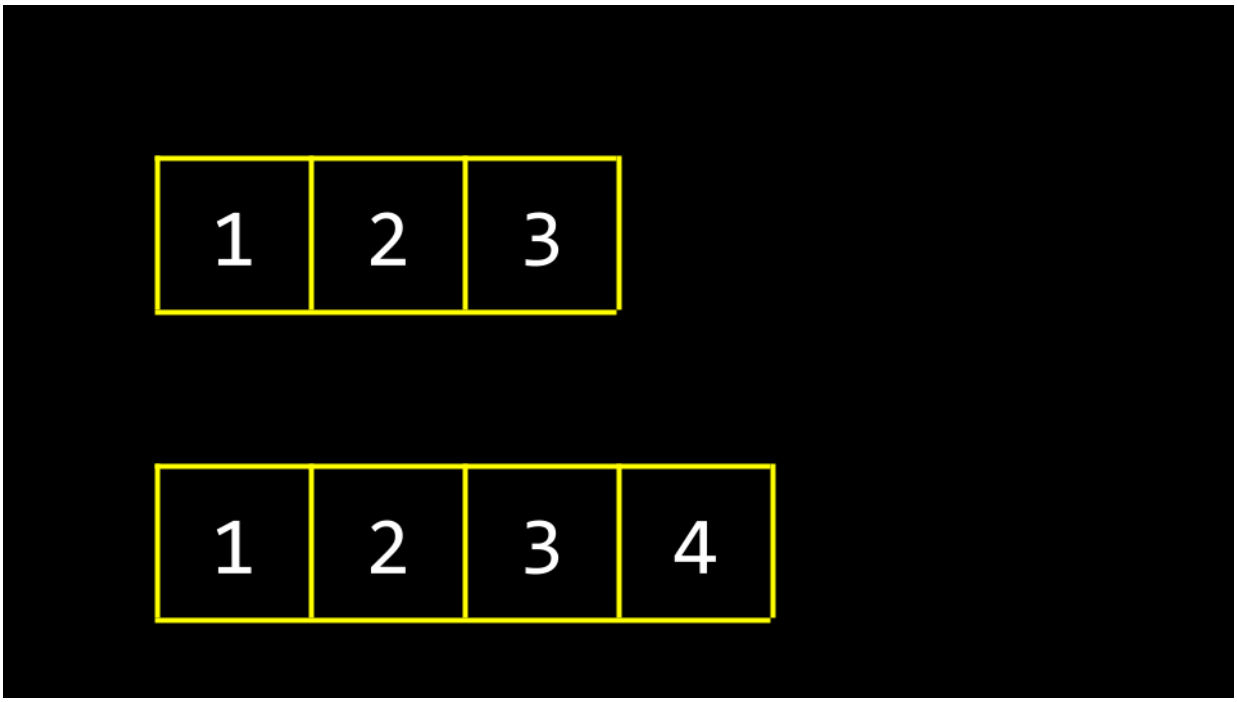
- Imagine que você queira armazenar um quarto valor `4` em nosso array. O que seria necessário é alocar uma nova área de memória e mover o array antigo para essa nova área? Inicialmente, essa nova área de memória seria preenchida com valores aleatórios.



- À medida que valores são adicionados a essa nova área de memória, os valores antigos e inválidos seriam sobrescritos.



- Eventualmente, todos os valores antigos e inválidos seriam sobrescritos com nossos novos dados.



- Uma das desvantagens dessa abordagem é que ela representa um design ruim: cada vez que adicionamos um número, temos que copiar o array item por item.

Matrizes

- Não seria ótimo se pudéssemos colocar isso em `4` outro lugar na memória? Por definição, isso não seria mais um array, pois `4` não estaria mais em memória contígua. Como poderíamos conectar diferentes locais na memória?
- No seu terminal, digite `code list.c` e escreva o código da seguinte forma:

```c
// Implements a list of numbers with an array of fixed size

#include <stdio.h>

int main(void)
{
    // List of size 3
    int list[3];

    // Initialize list with numbers
    list[0] = 1;
    list[1] = 2;
    list[2] = 3;

    // Print list
    for (int i = 0; i < 3; i++)
    {
        printf("%i\n", list[i]);
    }
}
```

Observe que o texto acima é muito semelhante ao que aprendemos anteriormente neste curso. A memória é pré-alocada para três itens.

- Com base no conhecimento adquirido recentemente, podemos aproveitar nossa compreensão de ponteiros para criar um design melhor neste código. Modifique seu código da seguinte forma:

```c
// Implements a list of numbers with an array of dynamic size

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

int main(void)
{
    // List of size 3
    int *list = malloc(3 * sizeof(int));
    if (list == NULL)
    {
        return 1;
    }

    // Initialize list of size 3 with numbers
    list[0] = 1;
    list[1] = 2;
    list[2] = 3;

    // List of size 4
    int *tmp = malloc(4 * sizeof(int));
    if (tmp == NULL)
    {
        free(list);
        return 1;
    }

    // Copy list of size 3 into list of size 4
    for (int i = 0; i < 3; i++)
    {
        tmp[i] = list[i];
    }

    // Add number to list of size 4
    tmp[3] = 4;

    // Free list of size 3
    free(list);

    // Remember list of size 4
    list = tmp;

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

    // Free list
    free(list);
    return 0;
}
```

Observe que uma lista de três inteiros é criada. Em seguida, três endereços de memória podem receber os valores `1`, `2`, e `3`. Depois, uma lista de quatro elementos é criada. A lista é então copiada da primeira para a segunda. O valor de `4` é adicionado à `tmp` lista. Como o bloco de memória `list` apontado por não está mais em uso, ele é liberado usando o comando `free(list)`. Finalmente, o compilador é instruído a apontar `list` o ponteiro para o bloco de memória `tmp` apontado por . O conteúdo de `list` é impresso e então liberado. Além disso, observe a inclusão de `stdlib.h`.

- É útil pensar em `x` `list` e ` `tmp` y` como sinais que apontam para um bloco de memória. Como no exemplo acima, `list` em um dado momento `x` *apontava* para um array de tamanho 3. Ao final, ` `list` y` passou a apontar para um bloco de memória de tamanho 4. Tecnicamente, ao final do código acima, `x` `tmp` e `list` `y` apontavam para o mesmo bloco de memória.
- Uma maneira de copiar o array sem usar um loop for é através de `realloc`:

```c
// Implements a list of numbers with an array of dynamic size using realloc

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

int main(void)
{
    // List of size 3
    int *list = malloc(3 * sizeof(int));
    if (list == NULL)
    {
        return 1;
    }

    // Initialize list of size 3 with numbers
    list[0] = 1;
    list[1] = 2;
    list[2] = 3;

    // Resize list to be of size 4
    int *tmp = realloc(list, 4 * sizeof(int));
    if (tmp == NULL)
    {
        free(list);
        return 1;
    }
    list = tmp;

    // Add number to list
    list[3] = 4;

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

    // Free list
    free(list);
```

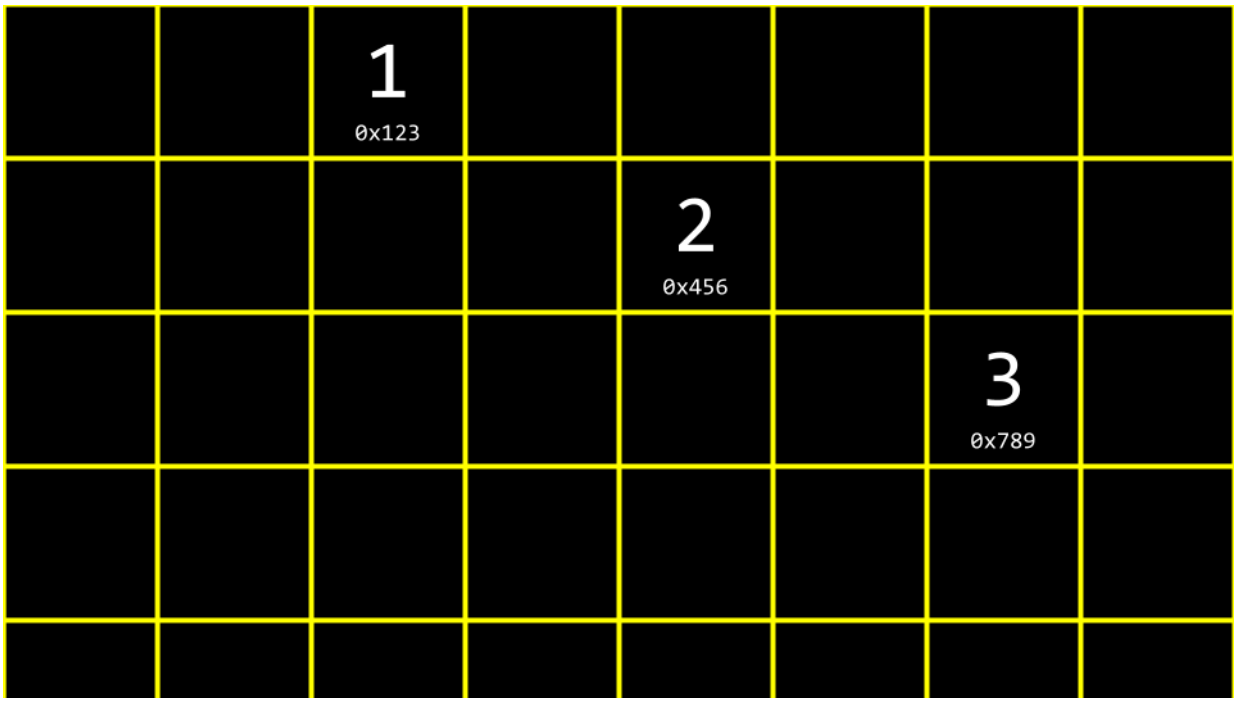
```
    return 0;
}
```

Observe que a lista é realocada para um novo array através de `realloc`.

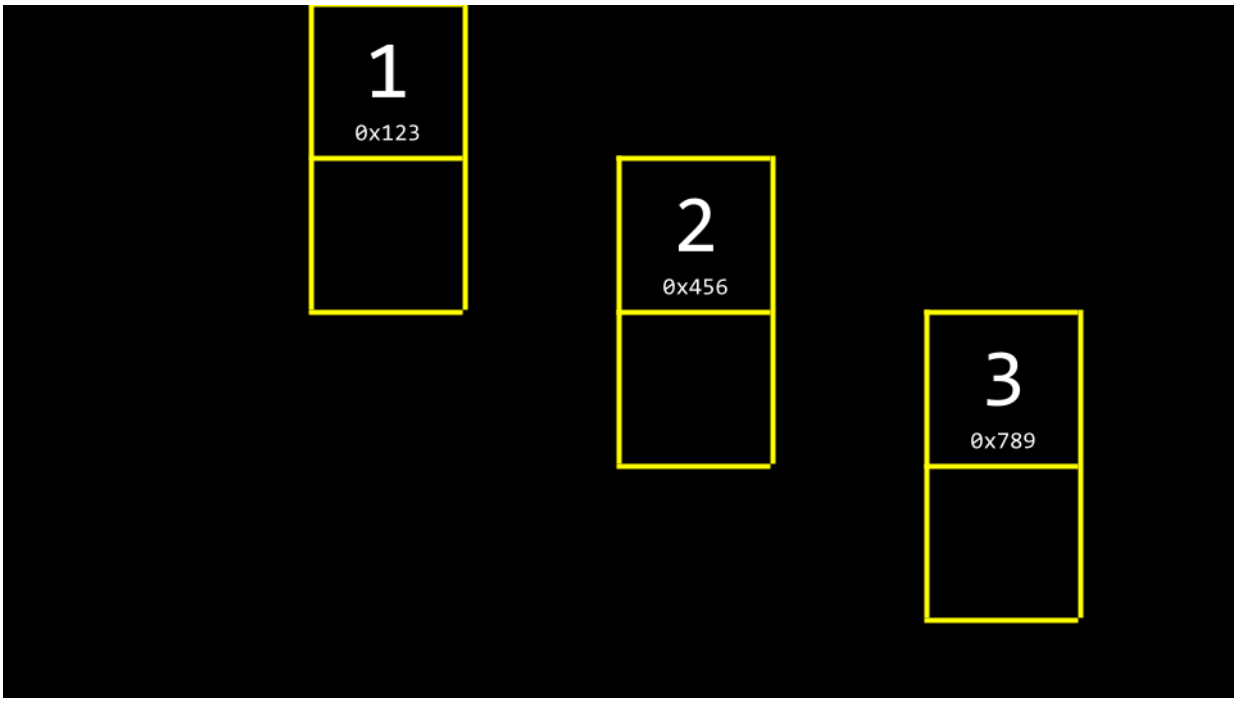
- Pode-se ter a tentação de alocar muito mais memória do que o necessário para a lista, como 30 itens em vez dos 3 ou 4 necessários. No entanto, isso é uma má prática de design, pois sobrecarrega os recursos do sistema quando eles não são potencialmente necessários. Além disso, há pouca garantia de que a memória para mais de 30 itens será necessária eventualmente.

Listas encadeadas

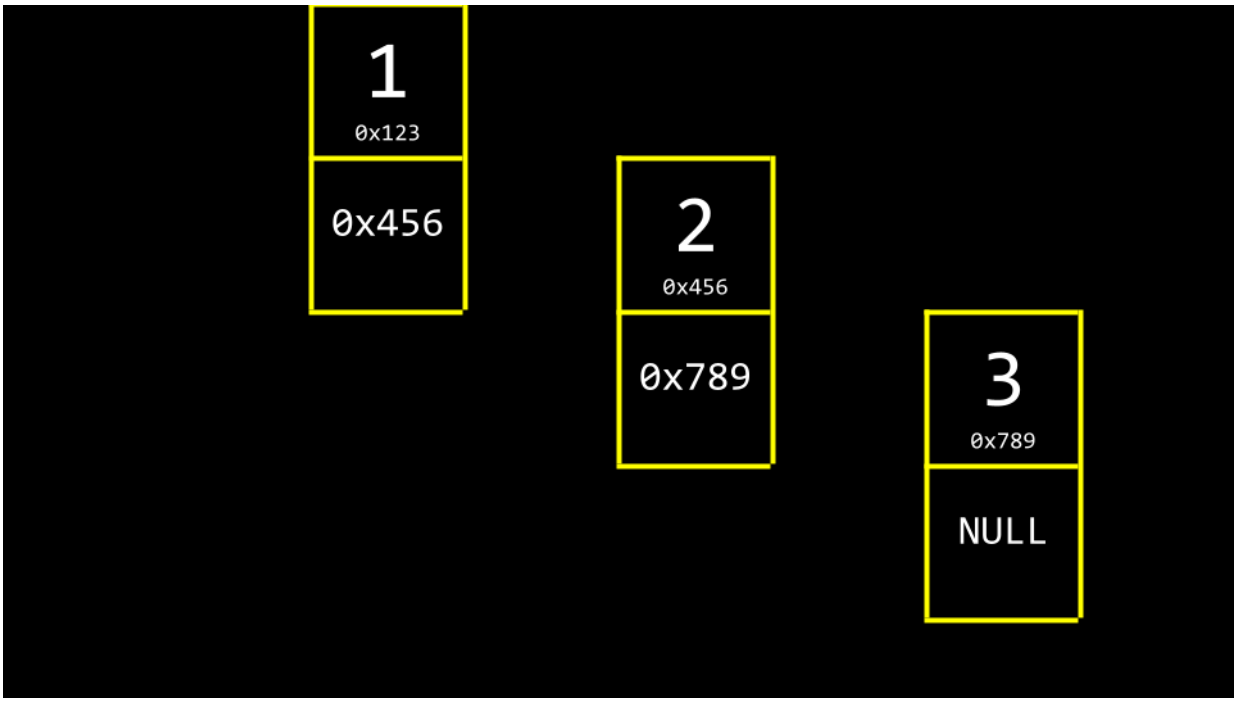
- Nas últimas semanas, você aprendeu sobre três tipos primitivos úteis. Um `struct` tipo de dado que você pode definir. Um ponteiro, `.` na *notação de ponto,* permite acessar variáveis dentro dessa estrutura. O `*` operador `&` é usado para declarar um ponteiro ou desreferenciar uma variável.
- Hoje, você conhecerá o `->` operador. É uma seta. Esse operador vai até um endereço e examina o interior de uma estrutura.
- Uma *lista ligada* é uma das estruturas de dados mais poderosas em C. Ela permite incluir valores localizados em diferentes áreas da memória. Além disso, permite aumentar e diminuir o tamanho da lista dinamicamente, conforme desejado.
- Você pode imaginar três valores armazenados em três áreas diferentes da memória, da seguinte forma:



- Como seria possível reunir esses valores em uma lista?
- Podemos imaginar os dados representados acima da seguinte forma:

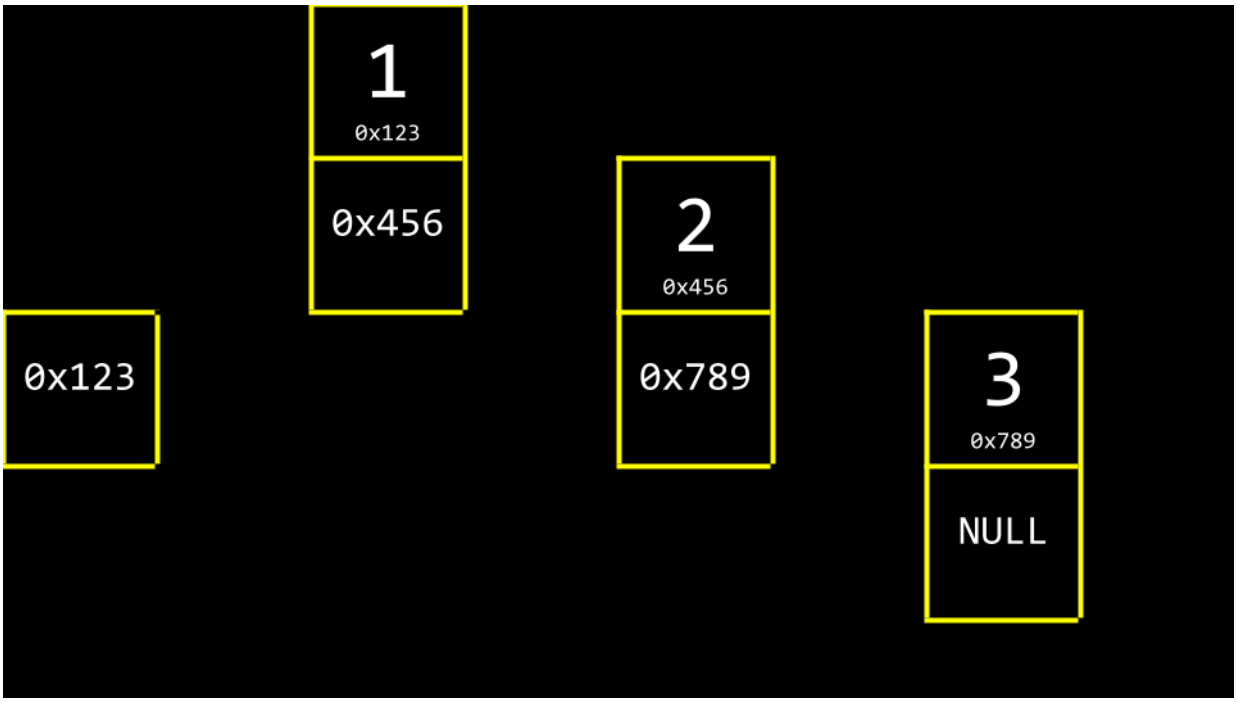


- Poderíamos utilizar mais memória para controlar a localização do próximo item usando um ponteiro.

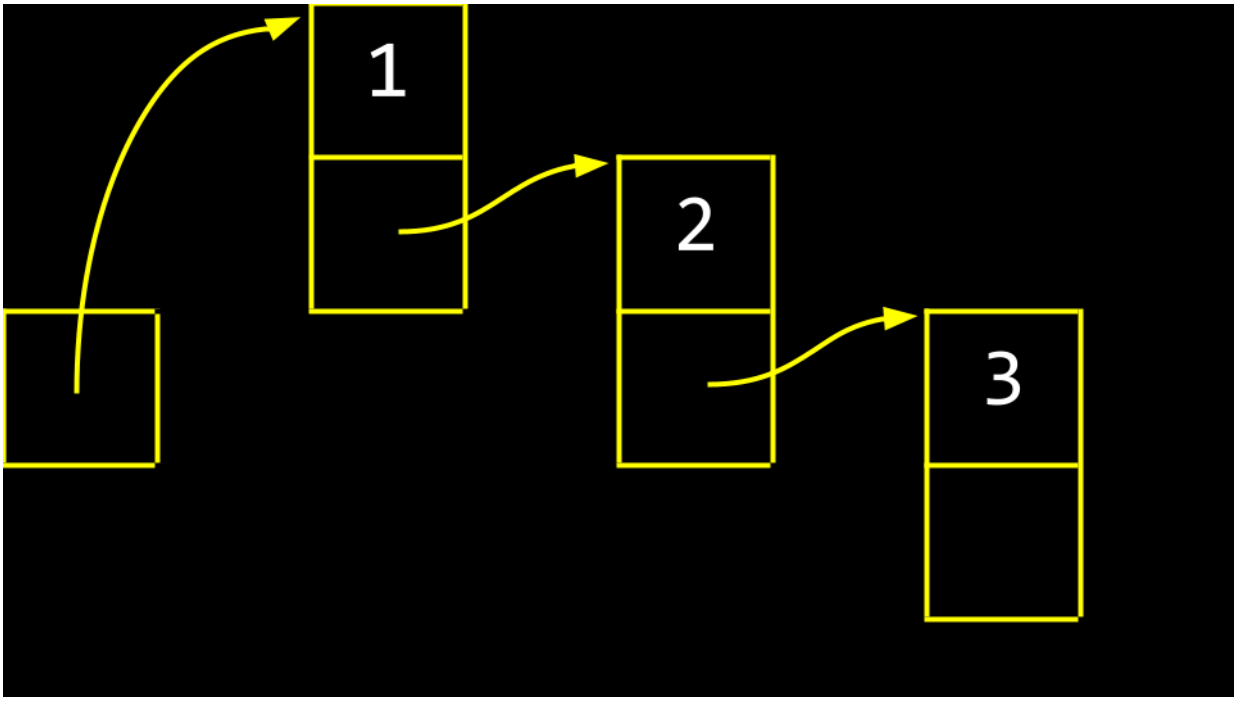


Observe que NULL é utilizado para indicar que não há mais nada *a seguir* na lista.

- Por convenção, manteríamos mais um elemento na memória, um ponteiro, que controla o primeiro item da lista, chamado de *cabeça* da lista.



- Abstraindo os endereços de memória, a lista ficaria assim:



- Essas caixas são chamadas *de nós* . Um *nó* contém um *item* e um ponteiro chamado *`next`* . Em código, você pode imaginar um nó da seguinte forma:

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

Observe que o item contido neste nó é um inteiro chamado `number` . Em segundo lugar, um ponteiro para um nó chamado `next` é incluído, o qual apontará para outro nó em algum lugar na memória.

- Podemos recriar isso `list.c` utilizando uma lista encadeada:

```c
// Start to build a linked list by prepending nodes

#include <cs50.h>
#include <stdio.h>
#include <stdlib.h>

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

int main(void)
{
    // Memory for numbers
    node *list = NULL;

    // Build list
    for (int i = 0; i < 3; i++)
    {
        // Allocate node for number
        node *n = malloc(sizeof(node));
        if (n == NULL)
        {
            return 1;
        }
        n->number = get_int("Number: ");
        n->next = NULL;

        // Prepend node to list
        n->next = list;
        list = n;
    }
    return 0;
}
```

Primeiro, um nó `node` é definido como um `struct` elemento da lista. Para cada elemento da lista, a memória para o nó `node` é alocada através `malloc` de um valor do tamanho de um nó. O campo `number` `n->number` (ou `next`) recebe um valor inteiro. O campo `next` (ou `next`) recebe um valor . Então, o nó é colocado no início da lista na posição de memória `next` . `n` `n->next` `n` `null` `list`

- Conceitualmente, podemos imaginar o processo de criação de uma lista encadeada. Primeiro, um elemento `node *list` é declarado, mas ele contém um valor inválido.

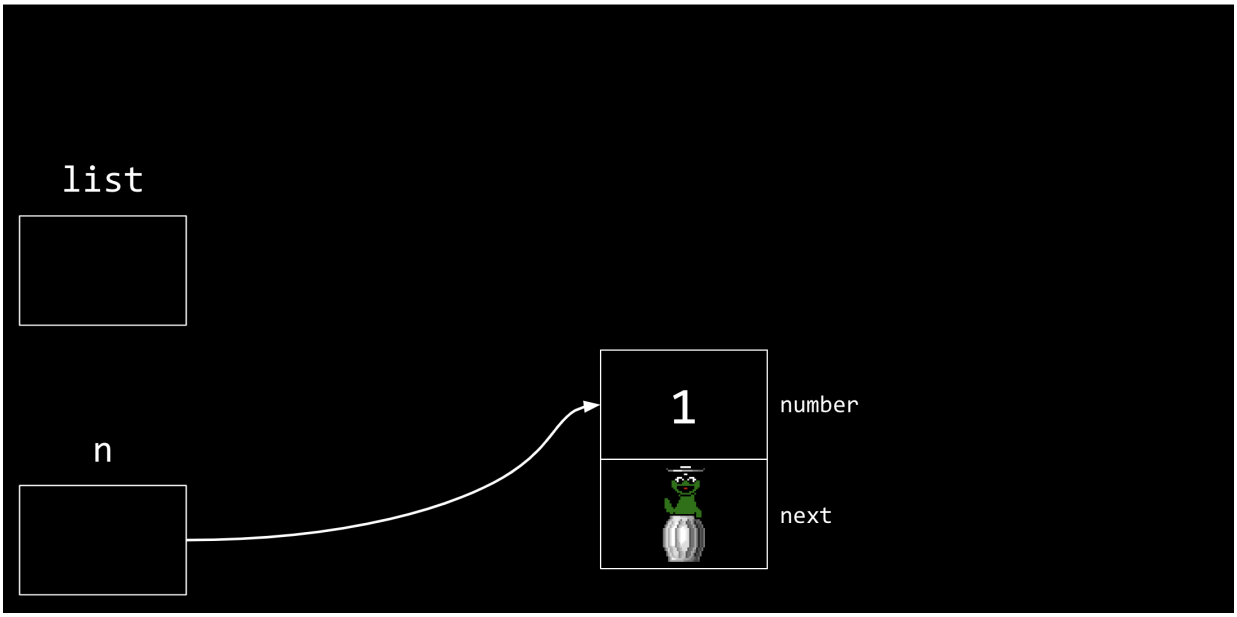
```
node *list;
```

- Em seguida, um nó chamado `n` é alocado na memória.

```
node *n = malloc(sizeof(node));
```

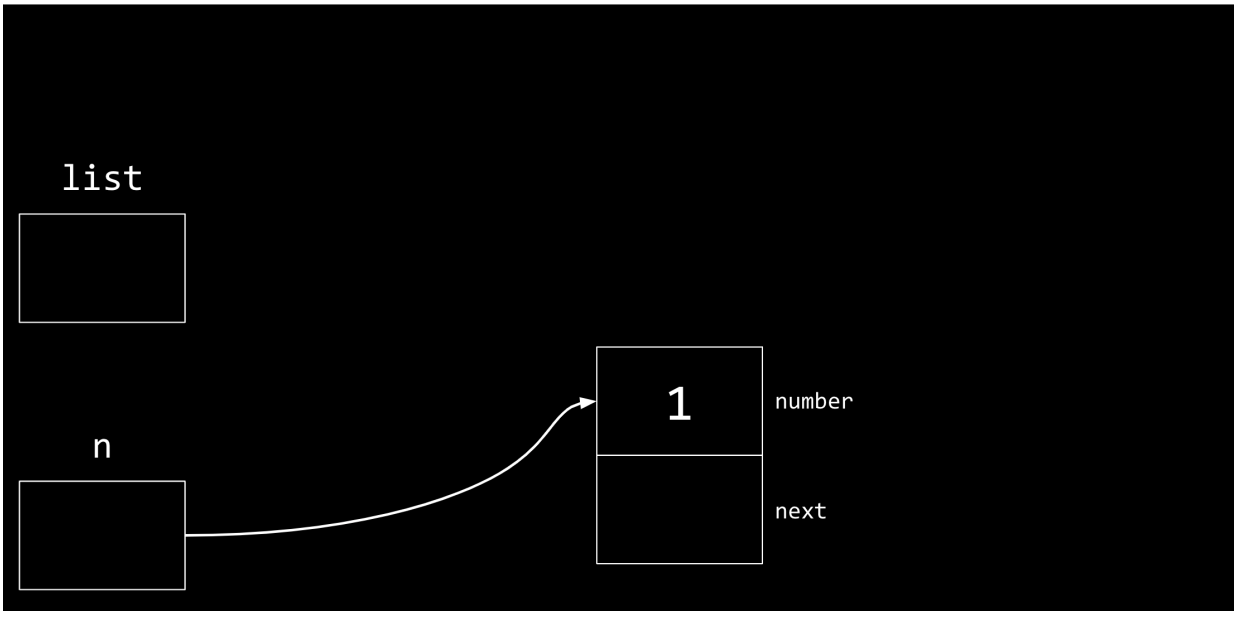
- Em seguida, o `number` valor é atribuído ao nó `1`.

```
n->number = 1;
```



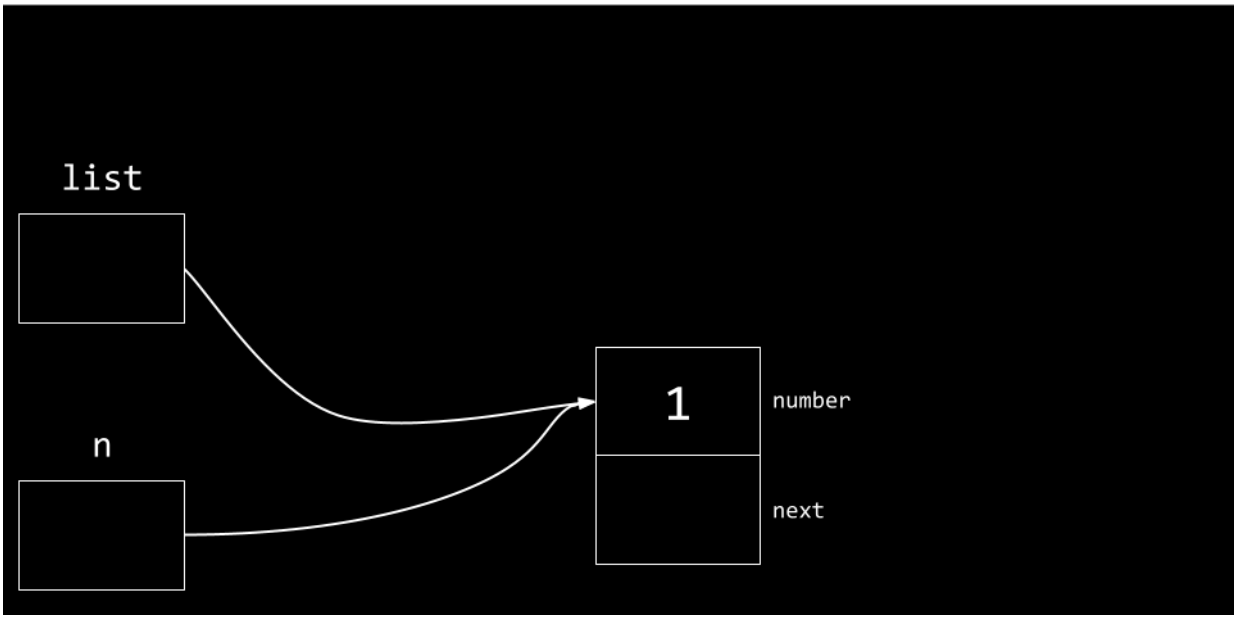
- Em seguida, o `next` campo do nó é atribuído `NULL`.

```
n->next = NULL;
```



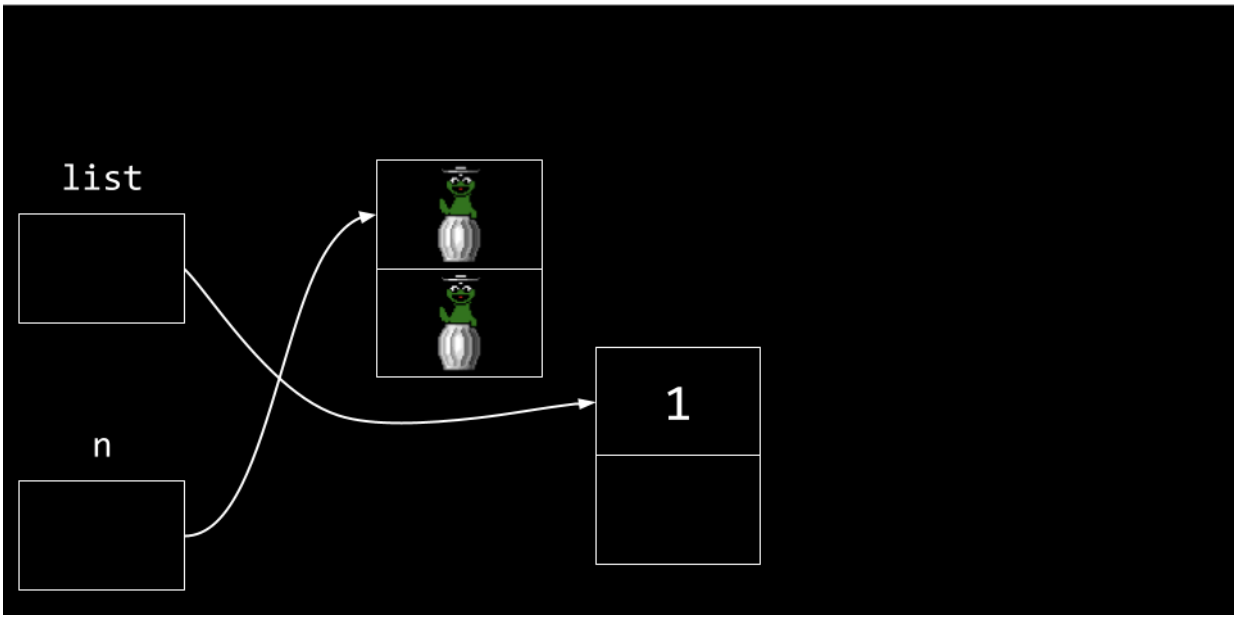
- Em seguida, `list` aponta para o local de memória para onde `n` aponta. `n` E `list` agora apontam para o mesmo local.

```
list = n;
```



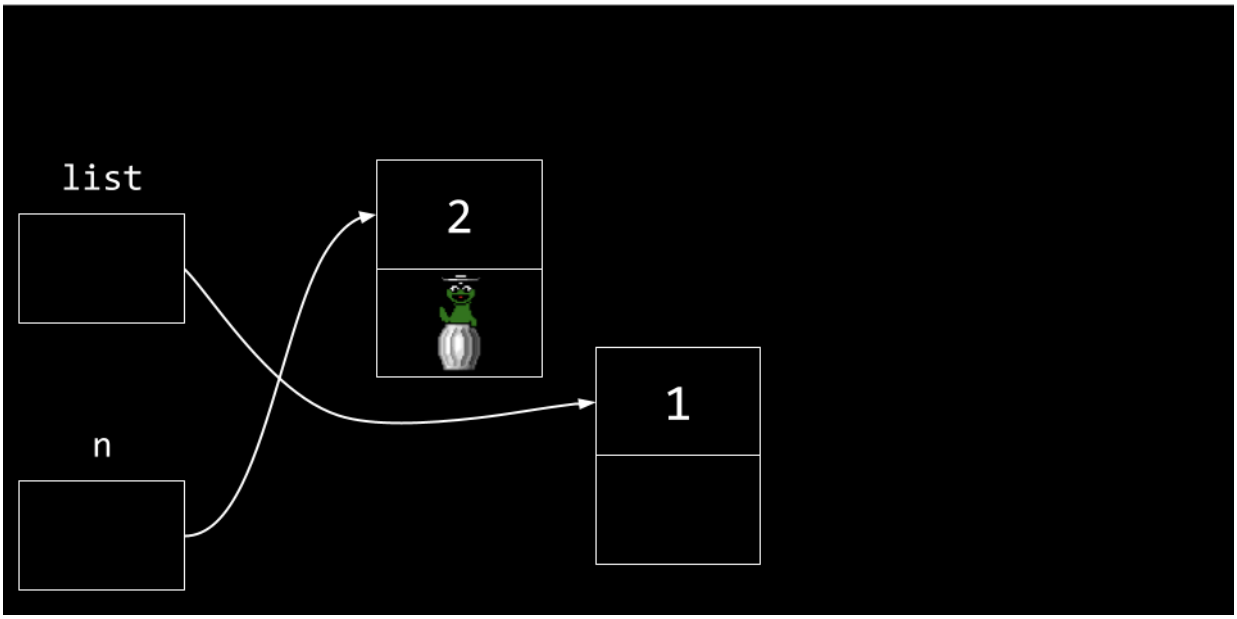
- Em seguida, um novo nó é criado. Tanto o campo ` `number` and` quanto `next` o campo `or` são preenchidos com valores inválidos.

```
node *n = malloc(sizeof(node));
```



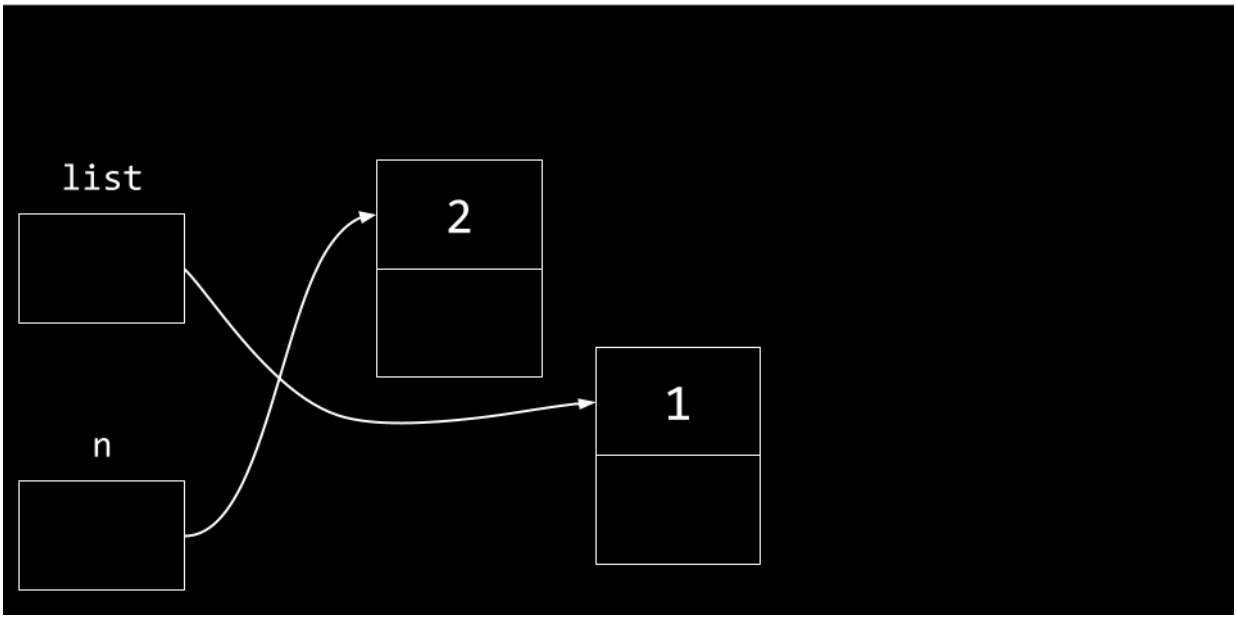
- O `number` valor do `n` nó 's (o novo nó) é atualizado para `2`.

```
n->number = 2;
```



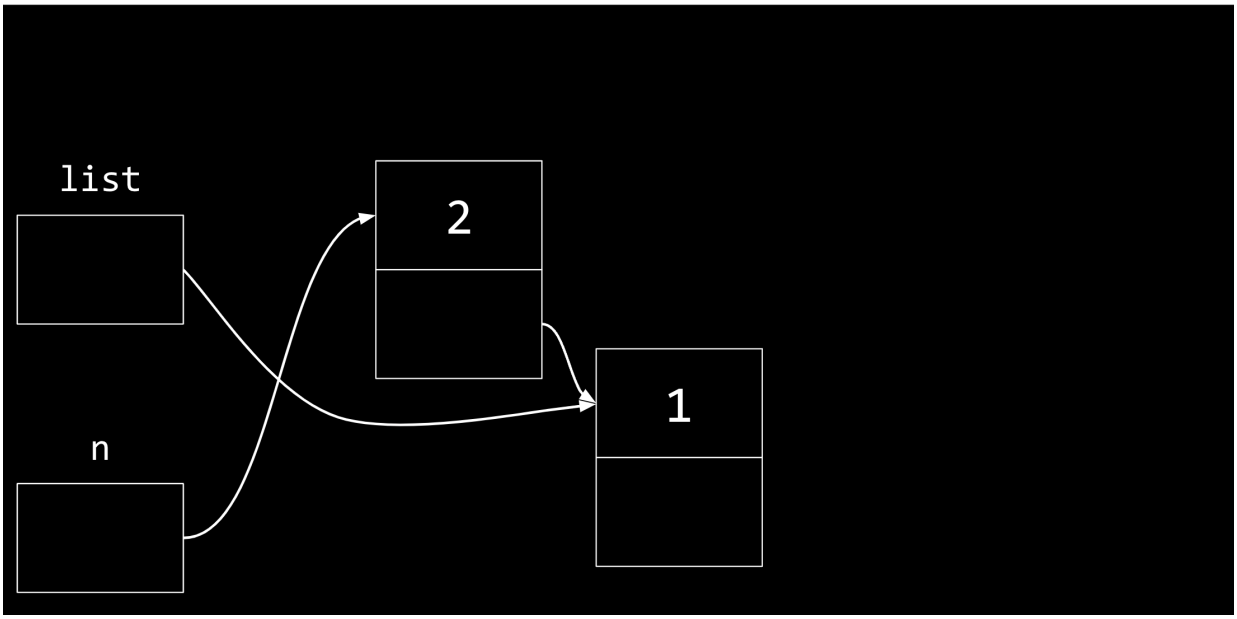
- Além disso, o `next` campo também foi atualizado.

```
n->next = NULL;
```



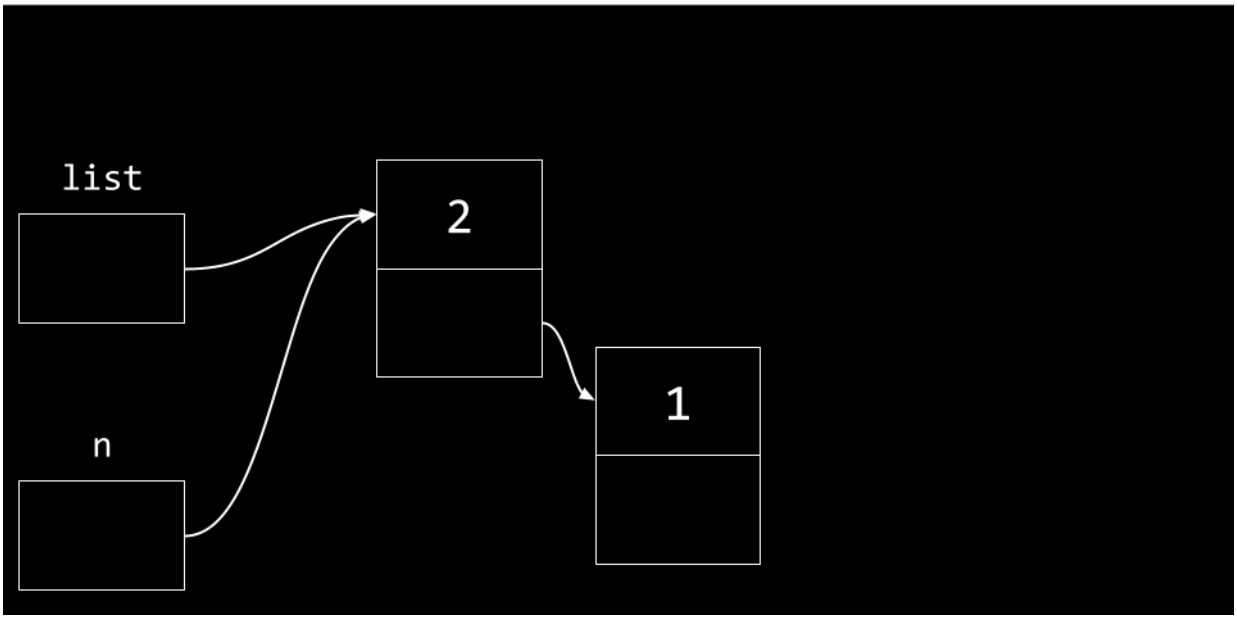
- Mais importante ainda, não queremos perder nossa conexão com nenhum desses nós, para que não se percam para sempre. Consequentemente, o campo `n` de 's `next` aponta para o mesmo local de memória que `list`.

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



- Finalmente, `list` é atualizado para apontar para `n`. Agora temos uma lista encadeada de dois itens.

```
list = n;
```



- Observando nosso diagrama da lista, podemos ver que o último número adicionado é o primeiro número que aparece na lista. Consequentemente, se imprimirmos a lista em ordem, começando pelo primeiro nó, a lista aparecerá fora de ordem.
- Podemos imprimir a lista na ordem correta da seguinte forma:

```c
// Print nodes in a linked list with a while loop

#include <cs50.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct node
{
    int number;
```

```
    struct node *next;
} node;

int main(void)
{
    // Memory for numbers
    node *list = NULL;

    // Build list
    for (int i = 0; i < 3; i++)
    {
        // Allocate node for number
        node *n = malloc(sizeof(node));
        if (n == NULL)
        {
            return 1;
        }
        n->number = get_int("Number: ");
        n->next = NULL;

        // Prepend node to list
        n->next = list;
        list = n;
    }

    // Print numbers
    node *ptr = list;
    while (ptr != NULL)
    {
        printf("%i\n", ptr->number);
        ptr = ptr->next;
    }
    return 0;
}
```

Observe que `x` `node *ptr = list` cria uma variável temporária que aponta para o mesmo local que ` `list` y` aponta. `x` `while` imprime o que o nó `ptr` aponta para `y` e, em seguida, atualiza `y` `ptr` para apontar para o `next` nó na lista.

- Neste exemplo, a inserção na lista ocorre sempre na ordem de$O(1)$, pois inserir um item no início de uma lista requer apenas um número muito pequeno de passos.
- Considerando o tempo necessário para pesquisar esta lista, ela está na seguinte ordem: $O(n)$Isso ocorre porque, no pior caso, a lista inteira precisa ser pesquisada para encontrar um item. A complexidade de tempo para adicionar um novo elemento à lista dependerá de onde esse elemento for adicionado. Isso é ilustrado nos exemplos abaixo.
- Listas encadeadas não são armazenadas em um bloco contíguo de memória. Elas podem crescer indefinidamente, desde que haja recursos de sistema suficientes. A desvantagem, no entanto, é que é necessária mais memória para controlar a lista em comparação com um array. Para cada elemento, é preciso armazenar não apenas o valor do elemento, mas também um ponteiro para o próximo nó. Além disso, listas encadeadas não podem ser indexadas como em um array, pois precisamos percorrer o primeiro elemento.n $- 1$elementos para encontrar a localização don$n$elemento. Por causa disso, a lista

mostrada acima deve ser pesquisada linearmente. Portanto, a busca binária não é possível em uma lista construída como a acima.

- Além disso, você pode adicionar números ao final da lista, como ilustrado neste código:

```
// Appends numbers to a linked list

#include <cs50.h>
#include <stdio.h>
#include <stdlib.h>

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

int main(void)
{
    // Memory for numbers
    node *list = NULL;

    // Build list
    for (int i = 0; i < 3; i++)
    {
        // Allocate node for number
        node *n = malloc(sizeof(node));
        if (n == NULL)
        {
            return 1;
        }
        n->number = get_int("Number: ");
        n->next = NULL;

        // If list is empty
        if (list == NULL)
        {
            // This node is the whole list
            list = n;
        }

        // If list has numbers already
        else
        {
            // Iterate over nodes in list
            for (node *ptr = list; ptr != NULL; ptr = ptr->next)
            {
                // If at end of list
                if (ptr->next == NULL)
                {
                    // Append node
                    ptr->next = n;
                    break;
                }
            }
        }
    }

    // Print numbers
```

```
    for (node *ptr = list; ptr != NULL; ptr = ptr->next)
    {
        printf("%i\n", ptr->number);
    }

    // Free memory
    node *ptr = list;
    while (ptr != NULL)
    {
        node *next = ptr->next;
        free(ptr);
        ptr = next;
    }
    return 0;
}
```

Observe como este código *percorre* a lista para encontrar o final. Ao adicionar um elemento (ao final da lista), nosso código será executado em $O(n)$ pois temos que percorrer toda a nossa lista antes de podermos adicionar o elemento final. Além disso, observe que uma variável temporária chamada `next` é usada para rastrear `ptr->next`.

- Além disso, você pode classificar sua lista à medida que os itens são adicionados:

```
// Implements a sorted linked list of numbers

#include <cs50.h>
#include <stdio.h>
#include <stdlib.h>

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

int main(void)
{
    // Memory for numbers
    node *list = NULL;

    // Build list
    for (int i = 0; i < 3; i++)
    {
        // Allocate node for number
        node *n = malloc(sizeof(node));
        if (n == NULL)
        {
            return 1;
        }
        n->number = get_int("Number: ");
        n->next = NULL;

        // If list is empty
        if (list == NULL)
        {
            list = n;
        }
```

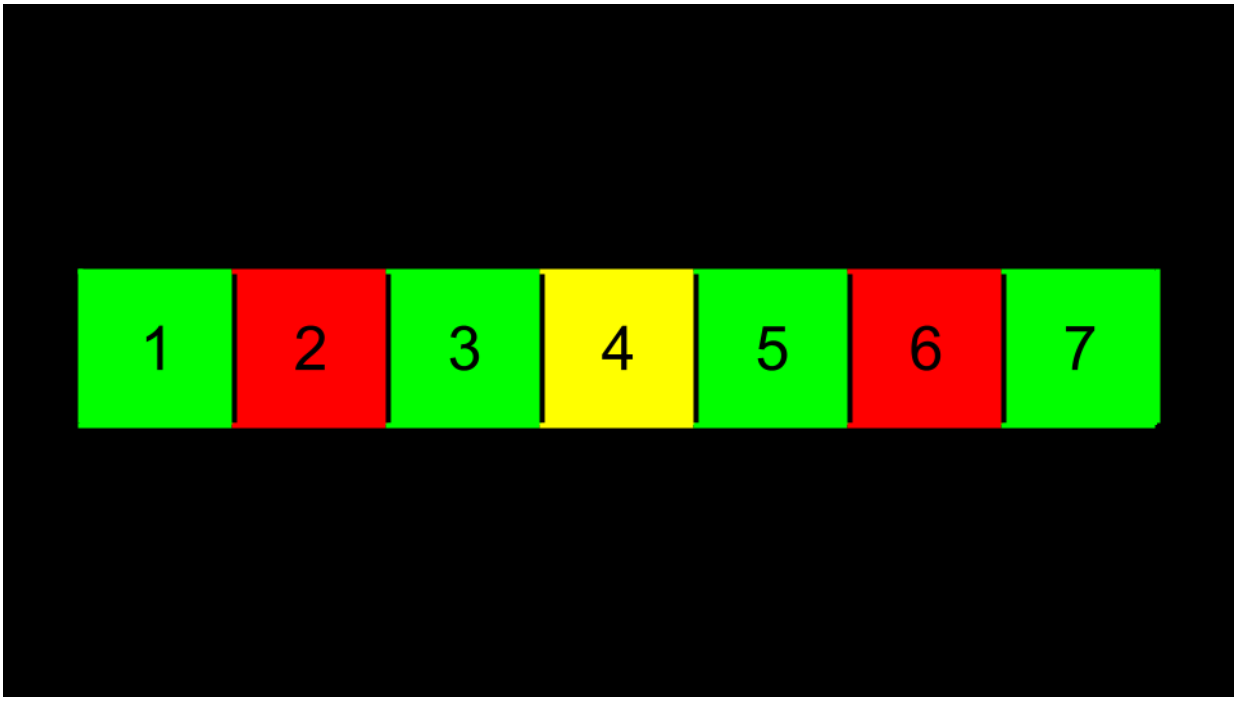
```
        // If number belongs at beginning of list
        else if (n->number < list->number)
        {
            n->next = list;
            list = n;
        }

        // If number belongs later in list
        else
        {
            // Iterate over nodes in list
            for (node *ptr = list; ptr != NULL; ptr = ptr->next)
            {
                // If at end of list
                if (ptr->next == NULL)
                {
                    // Append node
                    ptr->next = n;
                    break;
                }

                // If in middle of list
                if (n->number < ptr->next->number)
                {
                    n->next = ptr->next;
                    ptr->next = n;
                    break;
                }
            }
        }
    }

    // Print numbers
    for (node *ptr = list; ptr != NULL; ptr = ptr->next)
    {
        printf("%i\n", ptr->number);
    }

    // Free memory
    node *ptr = list;
    while (ptr != NULL)
    {
        node *next = ptr->next;
        free(ptr);
        ptr = next;
    }
    return 0;
}
```

Observe como esta lista é ordenada à medida que é construída. Para inserir um elemento nesta ordem específica, nosso código ainda será executado em$O(n)$Para cada inserção, na pior das hipóteses teremos que analisar todos os elementos existentes.

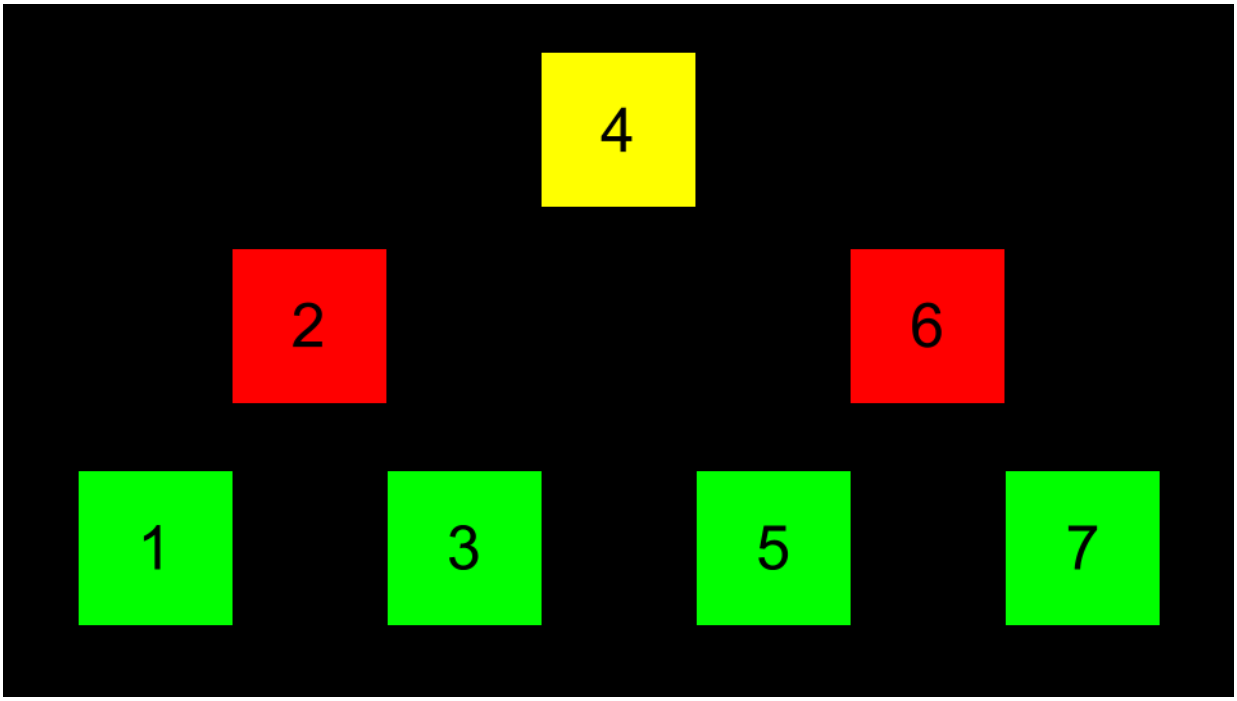
- Este código pode parecer complicado. No entanto, observe que, com ponteiros e a sintaxe acima, podemos combinar dados em diferentes locais da memória.

Árvores

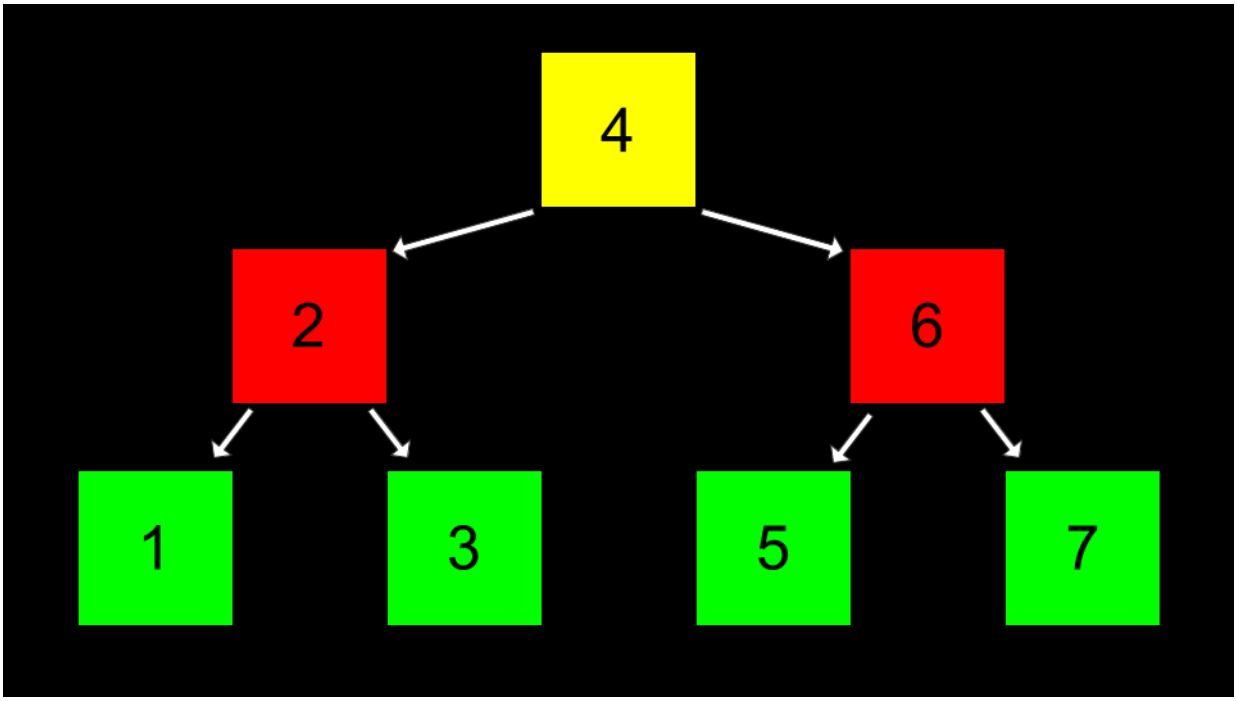
- Os arrays oferecem memória contígua que pode ser pesquisada rapidamente. Os arrays também oferecem a oportunidade de realizar buscas binárias.
- Será que poderíamos combinar o melhor dos arrays e das listas encadeadas?
- *Árvores de busca binária* são outra estrutura de dados que pode ser usada para armazenar dados de forma mais eficiente, permitindo que sejam pesquisados e recuperados.
- Você pode imaginar uma sequência ordenada de números.



- Imagine então que o valor central se torne o topo de uma árvore. Os valores menores que esse são colocados à esquerda. Os valores maiores que esse são colocados à direita.



- Os ponteiros podem então ser usados para apontar para a localização correta de cada área de memória, de forma que cada um desses nós possa ser conectado.



- Em código, isso pode ser implementado da seguinte forma.

```c
// Implements a list of numbers as a binary search tree

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

// Represents a node
typedef struct node
{
    int number;
    struct node *left;
    struct node *right;
}
node;

void free_tree(node *root);
void print_tree(node *root);

int main(void)
{
    // Tree of size 0
    node *tree = NULL;

    // Add number to list
    node *n = malloc(sizeof(node));
    if (n == NULL)
    {
        return 1;
    }
    n->number = 2;
    n->left = NULL;
    n->right = NULL;
    tree = n;

    // Add number to list
```

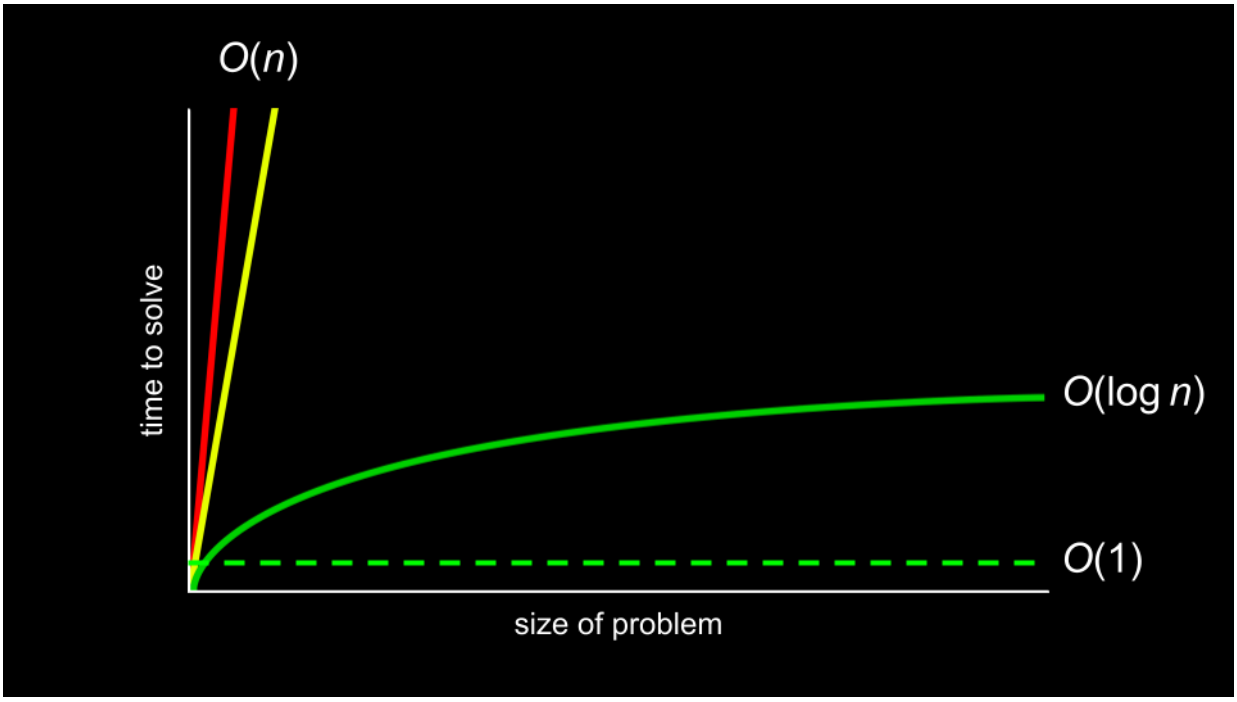
```
    n = malloc(sizeof(node));
    if (n == NULL)
    {
        free_tree(tree);
        return 1;
    }
    n->number = 1;
    n->left = NULL;
    n->right = NULL;
    tree->left = n;

    // Add number to list
    n = malloc(sizeof(node));
    if (n == NULL)
    {
        free_tree(tree);
        return 1;
    }
    n->number = 3;
    n->left = NULL;
    n->right = NULL;
    tree->right = n;

    // Print tree
    print_tree(tree);

    // Free tree
    free_tree(tree);
    return 0;
}

void free_tree(node *root)
{
    if (root == NULL)
    {
        return;
    }
    free_tree(root->left);
    free_tree(root->right);
    free(root);
}

void print_tree(node *root)
{
    if (root == NULL)
    {
        return;
    }
    print_tree(root->left);
    printf("%i\n", root->number);
    print_tree(root->right);
}
```

Observe que esta função de busca começa indo para a localização de `tree`. Em seguida, ela usa recursão para buscar `number`. A `free_tree` função libera a árvore recursivamente. `print_tree` imprime a árvore recursivamente.

- Uma árvore como a acima oferece um dinamismo que uma matriz não oferece. Ela pode crescer e encolher conforme desejarmos.
- Além disso, essa estrutura oferece um tempo de busca de $O(log n)$ quando a árvore estiver equilibrada.

Dicionários

- *Dicionários* são outra estrutura de dados.
- Dicionários, como os dicionários impressos que contêm uma palavra e uma definição, possuem uma *chave* e um *valor* .
- O *Santo Graal* da complexidade temporal algorítmica é $O(1)$ ou *tempo constante* . Isto é, o ideal é que o acesso seja instantâneo.



- Os dicionários podem oferecer essa velocidade de acesso por meio de hashing.

Hashing e tabelas hash

- *Hashing* é a ideia de pegar um valor e ser capaz de gerar um valor que se torne um atalho para ele posteriormente.
- Por exemplo, o hash de *"apple"* pode ser representado como um valor de 1 `1`, e *o de "berry"* como 1. `2` Portanto, encontrar *"apple"* é tão fácil quanto perguntar ao algoritmo *de hash onde "apple"* está armazenado. Embora não seja o ideal em termos de design, agrupar todos os valores *"a* " em um mesmo grupo e todos os valores *"b* " em outro, esse conceito de *agrupamento* de valores de hash ilustra como você pode usar esse conceito: um valor de hash pode ser usado para facilitar a busca por esse valor.
- Uma *função hash* é um algoritmo que reduz um valor grande a algo pequeno e previsível. Geralmente, essa função recebe um item que você deseja adicionar à sua tabela hash e

retorna um inteiro representando o índice do array no qual o item deve ser colocado.

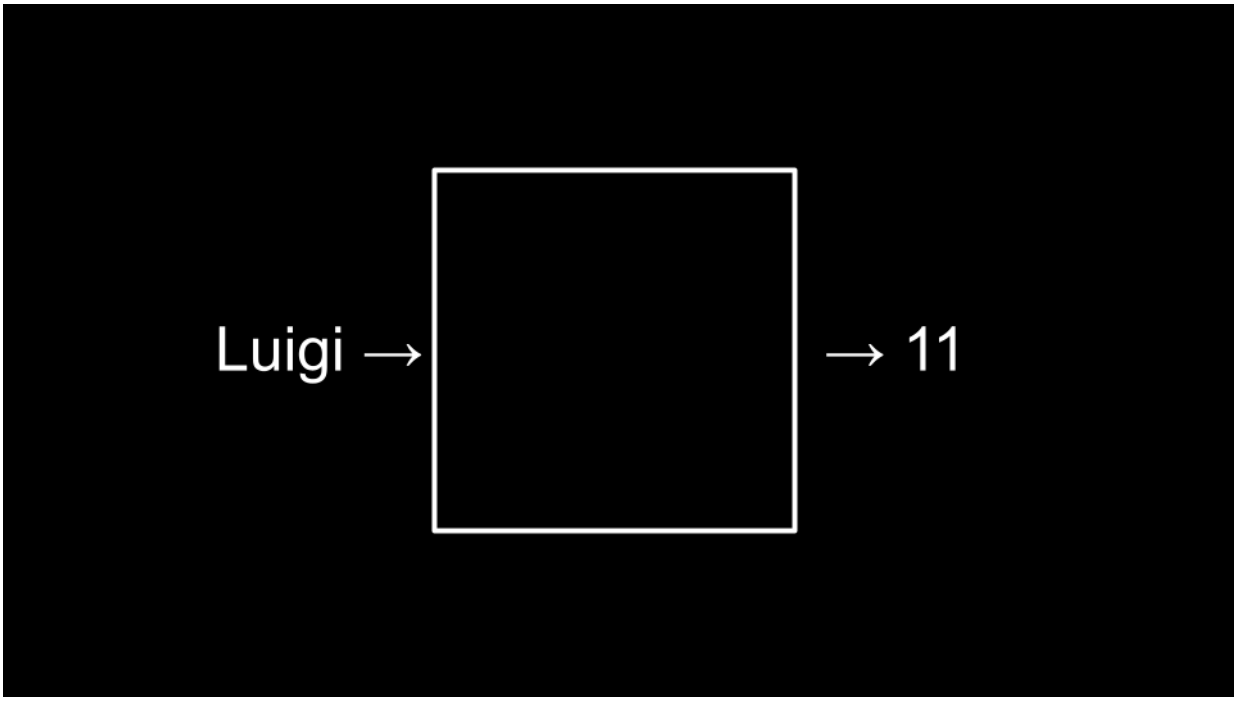
- Uma *tabela hash* é uma combinação fantástica de arrays e listas encadeadas. Quando implementada em código, uma tabela hash é um *array* de *ponteiros* para *nós* .
- Uma tabela hash pode ser imaginada da seguinte forma:

Observe que se trata de uma matriz à qual é atribuído cada valor do alfabeto.

- Em seguida, em cada posição da matriz, uma lista encadeada é usada para rastrear cada valor armazenado ali:

- *Colisões* ocorrem quando você adiciona valores à tabela hash e já existe algo naquela posição. No exemplo acima, as colisões são simplesmente adicionadas ao final da lista.
- As colisões podem ser reduzidas através de uma melhor programação da tabela hash e do algoritmo hash. Imagine uma melhoria em relação ao exemplo acima da seguinte forma:

- Considere o seguinte exemplo de um algoritmo de hash:



- Isso poderia ser implementado em código da seguinte forma:

```c
#include <ctype.h>

unsigned int hash(const char *word)
{
    return toupper(word[0]) - 'A';
}
```

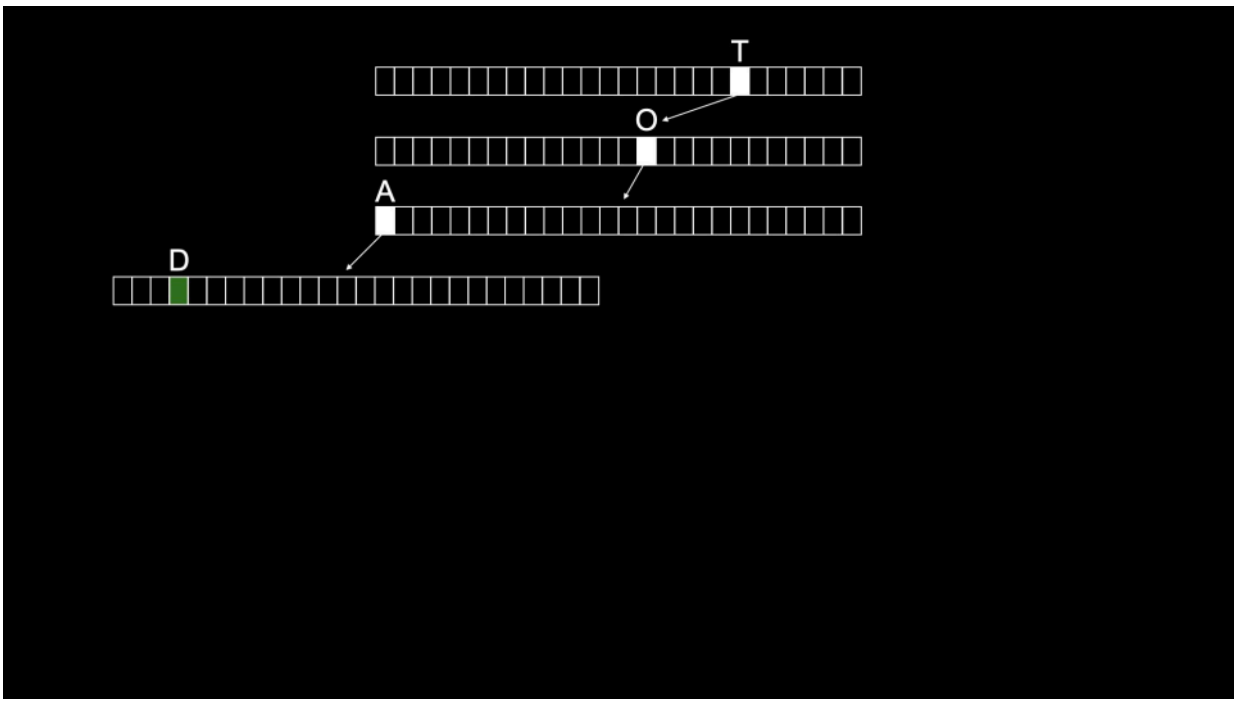
  Observe como a função hash retorna o valor de `toupper(word[0]) - 'A'`.

- Você, como programador, precisa decidir entre as vantagens de usar mais memória para ter uma tabela hash grande e potencialmente reduzir o tempo de busca, ou usar menos memória e potencialmente aumentar o tempo de busca.

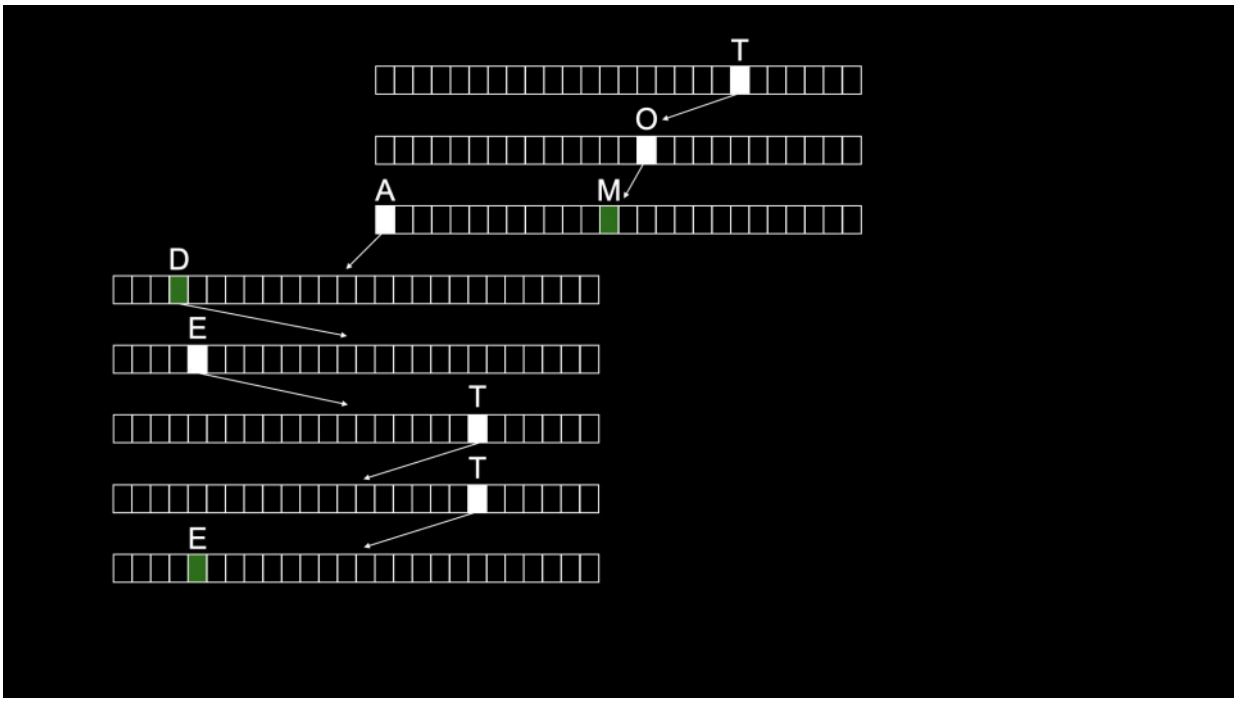
- Essa estrutura oferece um tempo de busca de$\mathrm{O(n)}$.

Tentativas

- *Tries* são outra forma de estrutura de dados. Tries são árvores de arrays.
- *As tentativas* são sempre pesquisáveis em tempo constante.
- Uma desvantagem dos *tries* é que eles tendem a ocupar muita memória. Observe que precisamos$26 \times 4 = 104$ `node` É só para guardar *o Sapo* !
- *O sapo* seria armazenado da seguinte forma:



- *Tom* seria então armazenado da seguinte forma:



- Essa estrutura oferece um tempo de busca de$\mathrm{O(1)}$.

- A desvantagem dessa estrutura é a quantidade de recursos necessários para utilizá-la.

Resumindo

Nesta lição, você aprendeu sobre como usar ponteiros para construir novas estruturas de dados. Especificamente, exploramos...

- Estruturas de dados
- Pilhas e filas
- Redimensionando matrizes
- Listas encadeadas
- Dicionários
- Tentativas

Até a próxima!