

Esto es CS50x

OpenCourseWare

Donar (<https://cs50.harvard.edu/donate>) [↗](https://community.alumni.harvard.edu/give/59206872) (<https://community.alumni.harvard.edu/give/59206872>)

David J. Malan (<https://cs.harvard.edu/malan/>)

malan@harvard.edu

















[f](https://www.facebook.com/dmalan) (<https://www.facebook.com/dmalan>) [G](https://github.com/dmalan) (<https://github.com/dmalan>) [@](https://www.instagram.com/davidjmalan/) (<https://www.instagram.com/davidjmalan/>) [in](https://www.linkedin.com/in/malan/) (<https://www.linkedin.com/in/malan/>) [ID](https://orcid.org/0000-0001-5338-2522) (<https://orcid.org/0000-0001-5338-2522>) [Q](https://www.quora.com/profile/David-J-Malan) (<https://www.quora.com/profile/David-J-Malan>) [R](https://www.reddit.com/user/davidjmalan) (<https://www.reddit.com/user/davidjmalan>) [T](https://twitter.com/davidjmalan) (<https://twitter.com/davidjmalan>)

Clase 5

- [Cambiar el tamaño de las matrices](#)
- [Estructuras de datos](#)
- [Listas vinculadas](#)
- [Implementando arreglos](#)
- [Implementando listas enlazadas](#)
- [Árboles](#)
- [Más estructuras de datos](#)

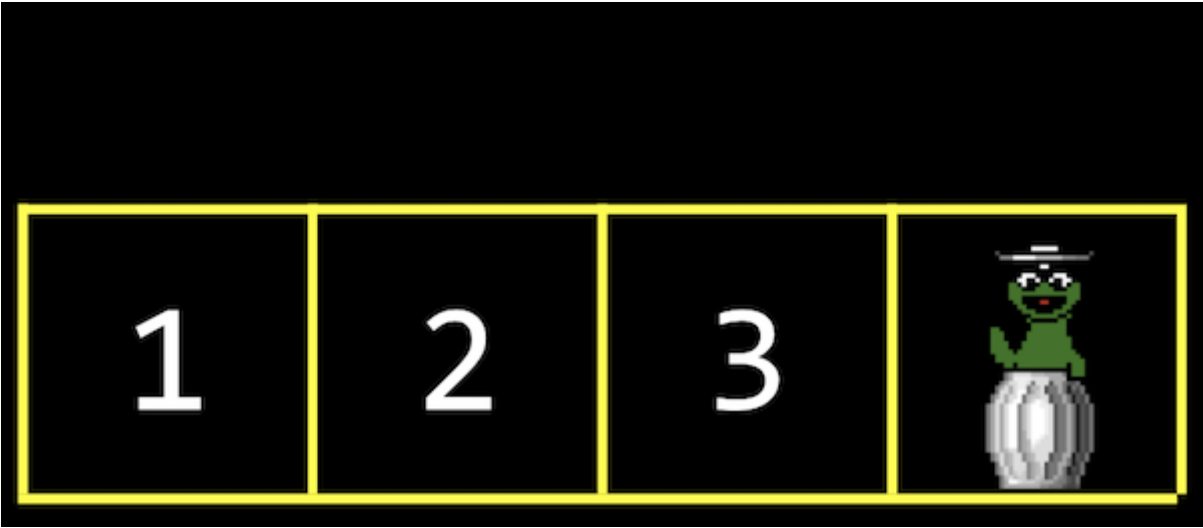
Cambiar el tamaño de las matrices

- La última vez, aprendimos sobre punteros `malloc` y otras herramientas útiles para trabajar con la memoria.
- En la semana 2, aprendimos sobre las matrices, donde podíamos almacenar el mismo tipo de valor en una lista, una tras otra en la memoria. Cuando necesitamos insertar un elemento, también necesitamos aumentar el tamaño de la matriz. Pero, es posible que la memoria posterior a ella en nuestra computadora ya se use para algunos otros datos, como una cadena:

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

- Una solución podría ser asignar más memoria donde haya suficiente espacio y mover nuestra matriz allí. Pero necesitaremos copiar nuestra matriz allí, que se convierte en una operación con un tiempo de ejecución de $O(n)$, ya que primero debemos copiar cada uno de los n elementos originales :

1	2	3
---	---	---



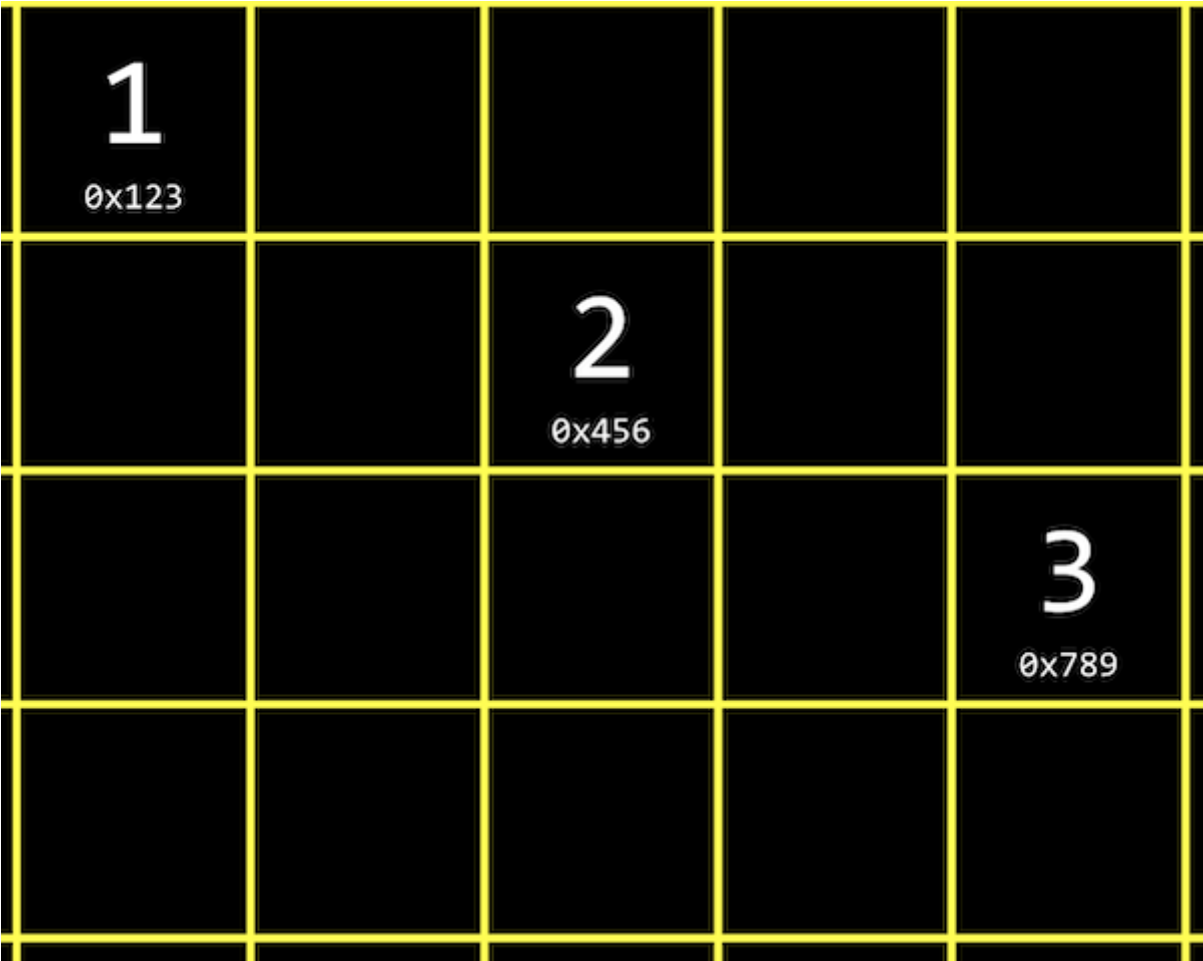
- El límite inferior de insertar un elemento en una matriz sería O (1) ya que es posible que ya tengamos espacio en la matriz para ello.

Estructuras de datos

- **Las estructuras de datos** son formas más complejas de organizar los datos en la memoria, lo que nos permite almacenar información en diferentes diseños.
- Para construir una estructura de datos, necesitaremos algunas herramientas:
 - `struct` para crear tipos de datos personalizados
 - `.` acceder a propiedades en una estructura
 - `*` para ir a una dirección en la memoria apuntada por un puntero
 - `->` para acceder a las propiedades en una estructura apuntada por un puntero

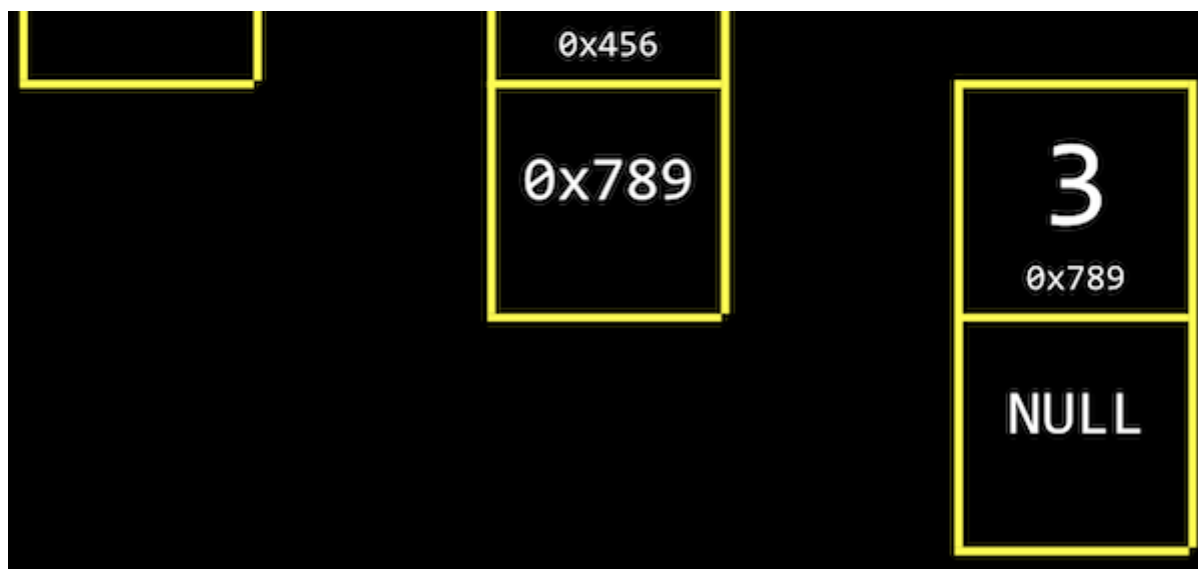
Listas vinculadas

- Con una **lista vinculada** , podemos almacenar una lista de valores que se pueden aumentar fácilmente almacenando valores en diferentes partes de la memoria:



- Tenemos los valores `1`, `2` y `3`, cada uno en alguna dirección en la memoria como `0x123`, `0x456`, y `0x789`.
- Esto es diferente a una matriz, ya que nuestros valores ya no están uno al lado del otro en la memoria. Podemos usar cualquier ubicación en la memoria que esté libre.
- Para rastrear todos estos valores, necesitamos vincular nuestra lista asignando, para cada elemento, suficiente memoria para el valor que queremos almacenar y la dirección del siguiente elemento:





- Junto a nuestro valor de `1`, por ejemplo, también almacenamos un puntero `0x456`, al siguiente valor. A esto lo llamaremos **nodo**, un componente de nuestra estructura de datos que almacena tanto un valor como un puntero. En C, implementaremos nuestros nodos con una estructura.
- Para nuestro último nodo con valor `3`, tenemos el puntero nulo, ya que no hay un elemento siguiente. Cuando necesitamos insertar otro nodo, podemos cambiar ese único puntero nulo para que apunte a nuestro nuevo valor.
- Tenemos la compensación de tener que asignar el doble de memoria para cada elemento, con el fin de dedicar menos tiempo a agregar valores. Y ya no podemos usar la búsqueda binaria, ya que nuestros nodos pueden estar en cualquier lugar de la memoria. Solo podemos acceder a ellos siguiendo los indicadores, uno a la vez.
- En el código, podríamos crear nuestra propia estructura llamada `node`, y necesitamos almacenar tanto nuestro valor, una `int` llamada `number`, como un puntero a la siguiente `node`, llamada `next`:

```

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

```

- Comenzamos esta estructura con `typedef struct node` para que podamos referirnos a un `node` interior de nuestra estructura.
- Podemos construir una lista vinculada en código comenzando con nuestra estructura. En primer lugar, vamos a querer recordar una lista vacía, así que podemos usar el puntero nulo: `node *list = NULL;`.
- Para agregar un elemento, primero necesitaremos asignar algo de memoria para un nodo y establecer sus valores:

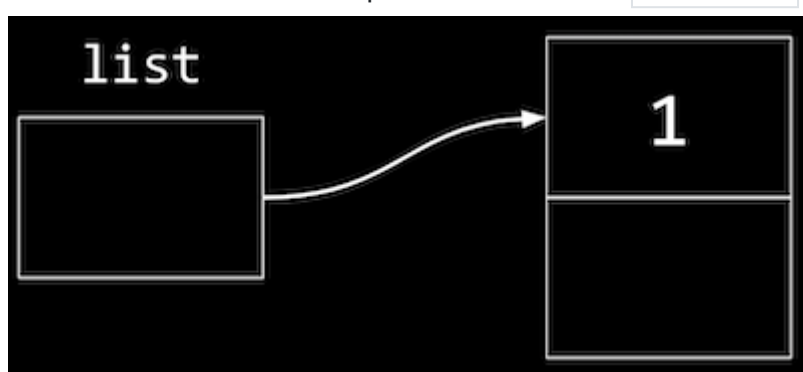
```

// We use sizeof(node) to get the right amount of memory to allocate, and
// malloc returns a pointer that we save as n
node *n = malloc(sizeof(node));

// We want to make sure malloc succeeded in getting memory for us
if (n != NULL)
{
    // This is equivalent to (*n).number, where we first go to the node pointed
    // to by n, and then set the number property. In C, we can also use this
    // arrow notation
    n->number = 1;
    // Then we need to make sure the pointer to the next node in our list
    // isn't a garbage value, but the new node won't point to anything (for now)
    n->next = NULL;
}

```

- Ahora nuestra lista debe apuntar a este nodo `list = n;`:



- Para agregar a la lista, crearemos un nuevo nodo de la misma manera asignando más memoria:

```

n = malloc(sizeof(node));
if (n != NULL)
{
    n->number = 2;
}

```

```
n->number = 2;
n->next = NULL;
}
```

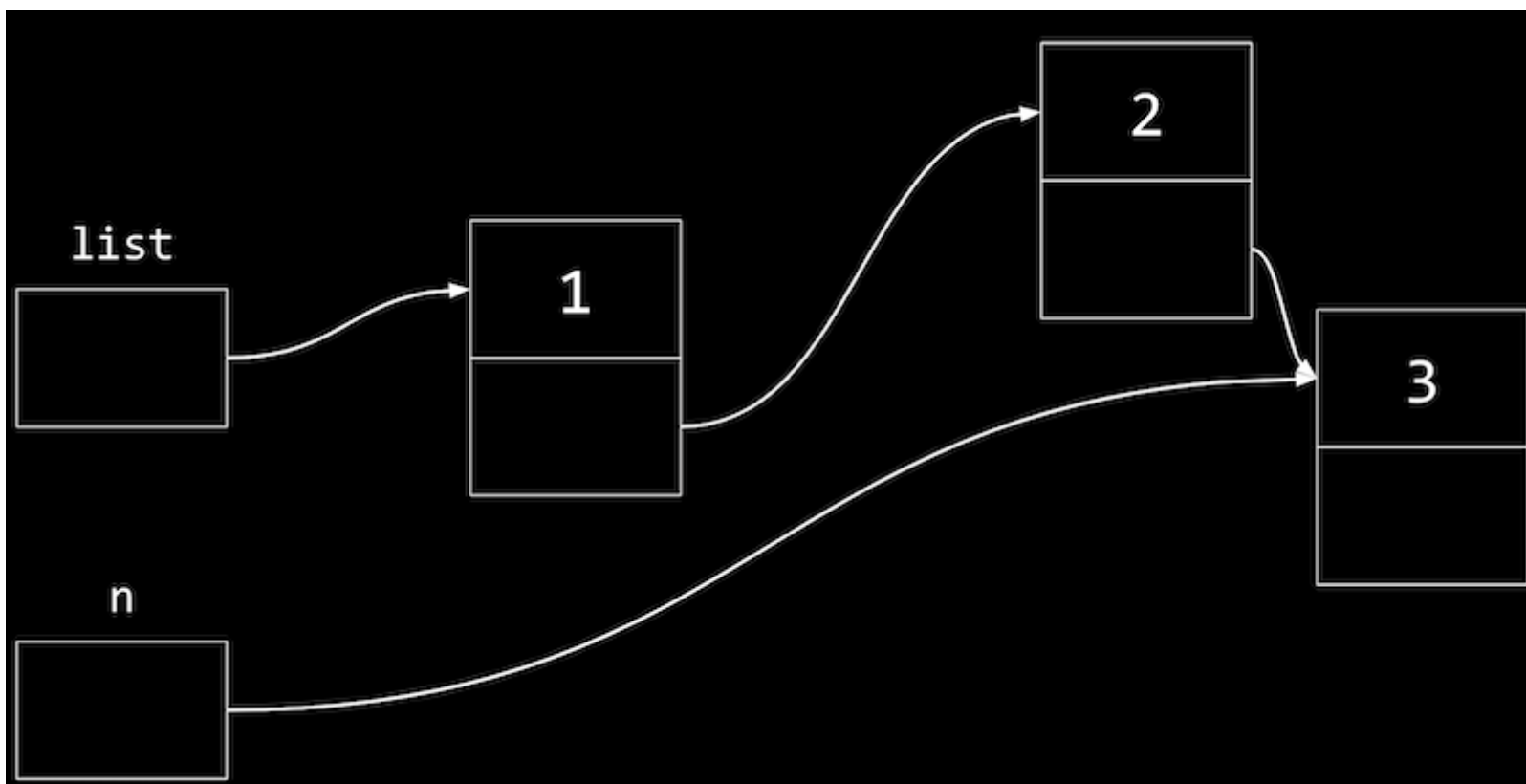
- Pero ahora necesitamos actualizar el puntero en nuestro primer nodo para que apunte a nuestro nuevo `n`:

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

- Para agregar un tercer nodo, haremos lo mismo siguiendo el `next` puntero en nuestra lista primero, luego estableciendo el `next` puntero *allí* para que apunte al nuevo nodo:

```
n = malloc(sizeof(node));
if (n != NULL)
{
    n->number = 3;
    n->next = NULL;
}
list->next->next = n;
```

- Gráficamente, nuestros nodos en la memoria se ven así:



- `n` es una variable temporal que apunta a nuestro nuevo nodo con valor 3.
- Queremos que el puntero en nuestro nodo con valor 2 apunte al nuevo nodo también, así que comenzamos desde `list` (que apunta al nodo con valor 1), seguimos el `next` puntero para llegar a nuestro nodo con valor 2 y actualizamos el `next` puntero. señalar `n`.
- Como resultado, la búsqueda en una lista vinculada también tendrá un tiempo de ejecución de $O(n)$, ya que necesitamos mirar todos los elementos en orden siguiendo cada puntero, incluso si la lista está ordenada. Insertar en una lista enlazada puede tener un tiempo de ejecución de $O(1)$, si insertamos nuevos nodos al principio de la lista.

Implementando arreglos

- Veamos cómo podemos implementar el cambio de tamaño de una matriz:

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

int main(void)
{
    // Use malloc to allocate enough space for an array with 3 integers
    int *list = malloc(3 * sizeof(int));
    if (list == NULL)
    {
        return 1;
    }

    // Set the values in our array
    list[0] = 1;
    list[1] = 2;
    list[2] = 3;

    // Now if we want to store another value, we can allocate more memory
    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 new number to list of size 4
    tmp[3] = 4;

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

    // Remember new list of size 4
    list = tmp;

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

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

- Recuerde que `malloc` asigna y libera memoria del área del montón. Resulta que podemos llamar a otra función de biblioteca `realloc`, para reasignar algo de memoria que asignamos anteriormente:

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

- Y `realloc` copia nuestra matriz anterior, `list` para nosotros en una porción más grande de memoria del tamaño que pasamos. Si sucede que hay espacio después de nuestra porción de memoria existente, obtendremos la misma dirección, pero con la memoria después. asignado a nuestra variable también.

Implementando listas enlazadas

- Combinemos nuestros fragmentos de código de antes en un programa que implemente una lista vinculada:

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

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

int main(void)
{
    // List of size 0. We initialize the value to NULL explicitly, so there's
    // no garbage value for our list variable
    node *list = NULL;

    // Allocate memory for a node, n
    node *n = malloc(sizeof(node));
    if (n == NULL)
    {
        return 1;
    }

    // Set the value and pointer in our node
    n->number = 1;
    n->next = NULL;

    // Add node n by pointing list to it, since we only have one node so far
    list = n;

    // Allocate memory for another node, and we can reuse our variable n to
    // point to it, since list points to the first node already
    n = malloc(sizeof(node));
    if (n == NULL)
    {
        free(list);
        return 1;
    }

    // Set the values in our new node
    n->number = 2;
    n->next = NULL;

    // Update the pointer in our first node to point to the second node
    list->next = n;

    // Allocate memory for a third node
    n = malloc(sizeof(node));
    if (n == NULL)
    {
        // Free both of our other nodes
        free(list->next);
        free(list);
        return 1;
    }
```

```

    return 1;
}
n->number = 3;
n->next = NULL;

// Follow the next pointer of the list to the second node, and update
// the next pointer there to point to n
list->next->next = n;

// Print list using a loop, by using a temporary variable, tmp, to point
// to list, the first node. Then, every time we go over the loop, we use
// tmp = tmp->next to update our temporary pointer to the next node. We
// keep going as long as tmp points to somewhere, stopping when we get to
// the last node and tmp->next is null.
for (node *tmp = list; tmp != NULL; tmp = tmp->next)
{
    printf("%i\n", tmp->number);
}

// Free list, by using a while loop and a temporary variable to point
// to the next node before freeing the current one
while (list != NULL)
{
    // We point to the next node first
    node *tmp = list->next;
    // Then, we can free the first node
    free(list);
    // Now we can set the list to point to the next node
    list = tmp;
    // If list is null, when there are no nodes left, our while loop will stop
}
}

```

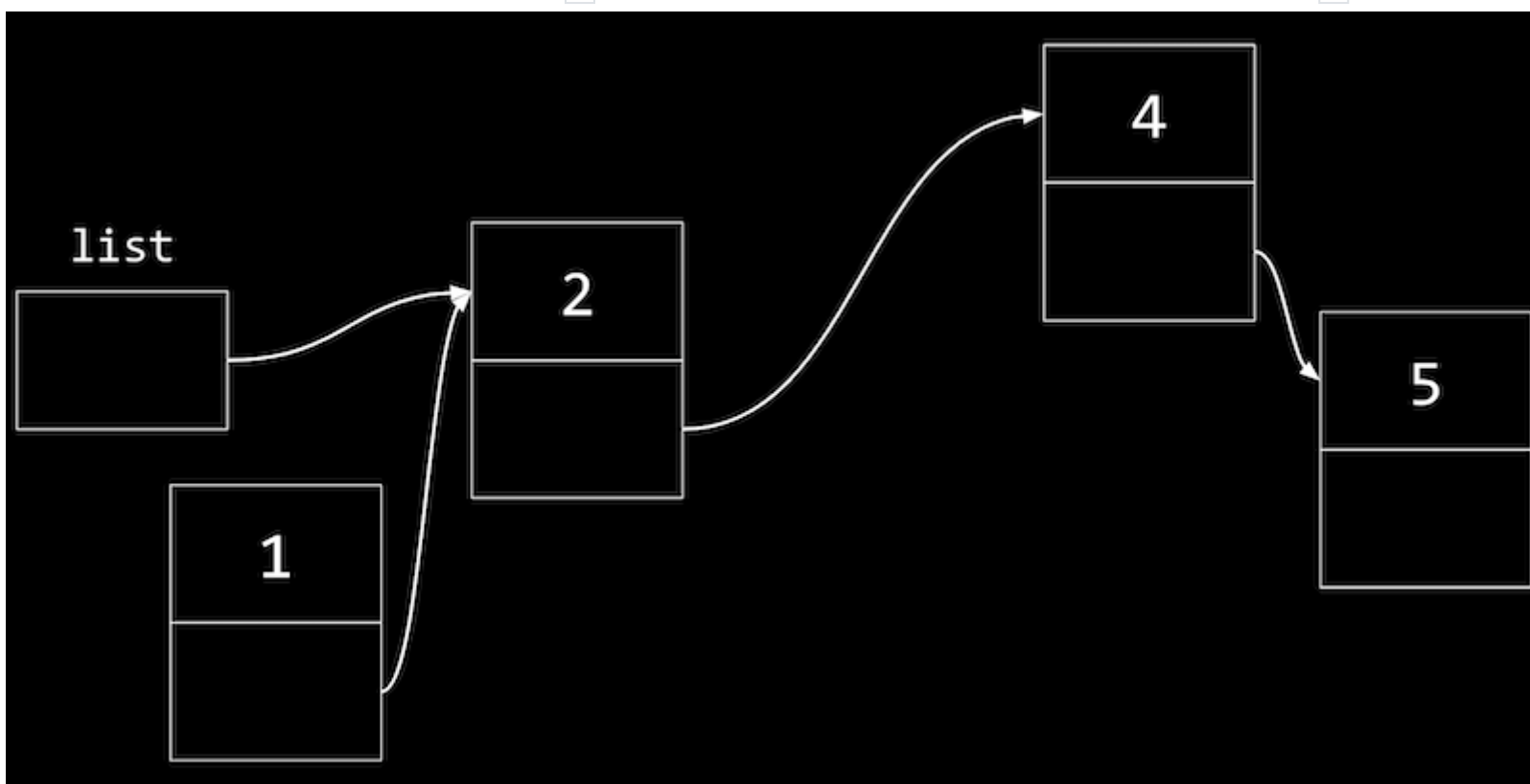
- Si queremos insertar un nodo al frente de nuestra lista vinculada, necesitaríamos actualizar cuidadosamente nuestro nodo para que apunte al siguiente, antes de actualizar la variable de la lista. De lo contrario, perderemos el resto de nuestra lista:

```

// Here, we're inserting a node into the front of the list, so we want its
// next pointer to point to the original list. Then we can change the list to
// point to n.
n->next = list;
list = n;

```

- Al principio, tendremos un nodo con valor 1 apuntando al inicio de nuestra lista, un nodo con valor 2:

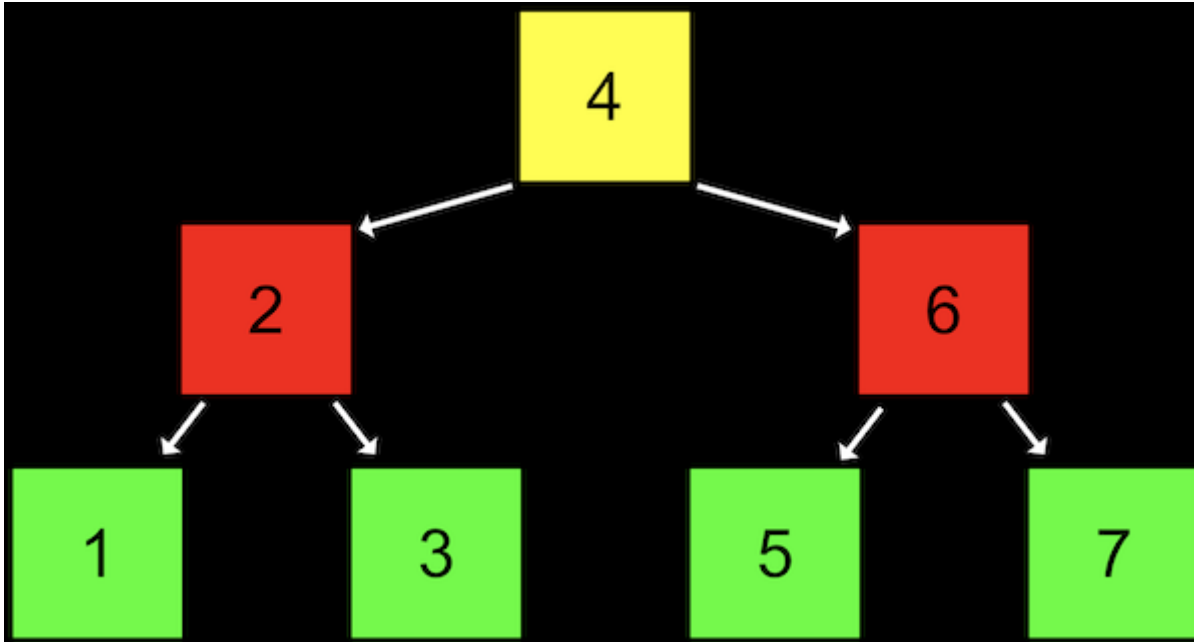


- Ahora podemos actualizar nuestra `list` variable para que apunte al nodo con valor 1 y no perder el resto de nuestra lista.
- De manera similar, para insertar un nodo en el medio de nuestra lista, `next` primero cambiamos el puntero del nuevo nodo para que apunte al resto de la lista, luego actualizamos el nodo anterior para que apunte al nuevo nodo.
- Una lista vinculada demuestra cómo podemos usar punteros para construir estructuras de datos flexibles en la memoria, aunque solo lo estamos visualizando en una dimensión.

- Con una matriz ordenada, podemos usar la búsqueda binaria para encontrar un elemento, comenzando en el medio (amarillo), luego en el medio de cualquier mitad (rojo) y finalmente a la izquierda o derecha (verde) según sea necesario:



- Con una matriz, podemos acceder aleatoriamente a elementos en $O(1)$ tiempo, ya que podemos usar aritmética para ir a un elemento en cualquier índice.
- Un **árbol** es otra estructura de datos donde cada nodo apunta a otros dos nodos, uno a la izquierda (con un valor menor) y otro a la derecha (con un valor mayor):



- Observe que ahora visualizamos esta estructura de datos en dos dimensiones (aunque los nodos en la memoria pueden estar en cualquier ubicación).
- Y podemos implementar esto con una versión más compleja de un nodo en una lista vinculada, donde cada nodo tiene no uno sino dos punteros a otros nodos. Todos los valores a la izquierda de un nodo son más pequeños y todos los valores de los nodos a la derecha son mayores, lo que permite utilizarlo como un **árbol de búsqueda binaria**. Y la estructura de datos se define en sí misma de forma recursiva, por lo que podemos utilizar funciones recursivas para trabajar con ella.
- Cada nodo tiene como máximo dos **hijos**, o nodos a los que apunta.
- Y como una lista enlazada, queremos mantener un puntero al principio de la lista, pero en este caso queremos apuntar a la **raíz** o al nodo central superior del árbol (el 4).
- Podemos definir un nodo con no uno sino dos punteros:

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

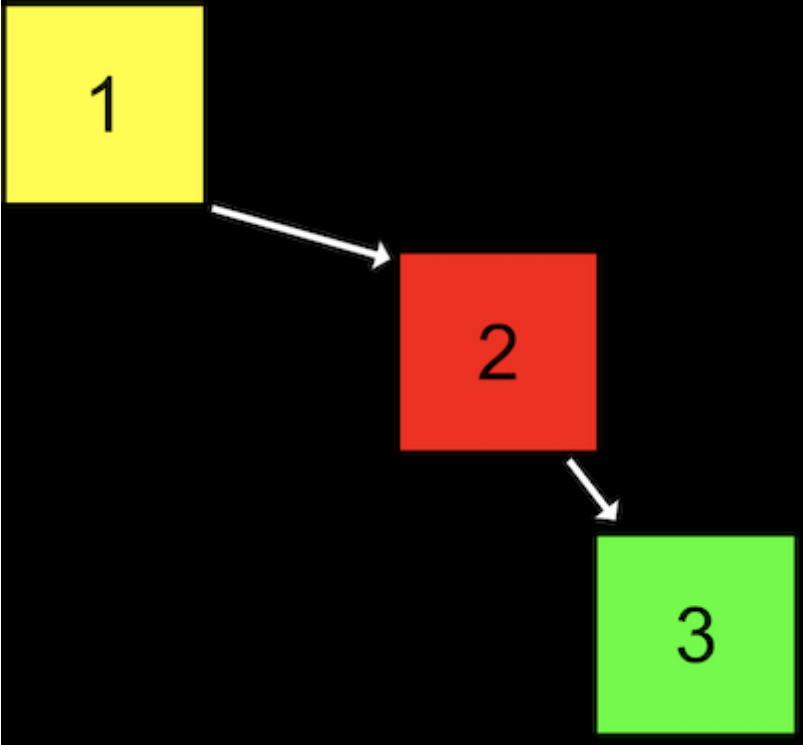
- Y escribe una función para buscar recursivamente un árbol:

```
// tree is a pointer to a node that is the root of the tree we're searching in.
// number is the value we're trying to find in the tree.
bool search(node *tree, int number)
{
    // First, we make sure that the tree isn't NULL, if we've reached a node
    // on the bottom, or if our tree is entirely empty
    if (tree == NULL)
    {
        return false;
    }
    // If we're looking for a number that's less than the tree's number,
    // search the left side, using the node on the left as the new root
    else if (number < tree->number)
    {
        return search(tree->left, number);
    }
    // If we're looking for a number that's greater than the tree's number,
    // search the right side, using the node on the right as the new root
    else if (number > tree->number)
    {
        return search(tree->right, number);
    }
    // If we've found the number, return true
    return true;
}
```



```
}
// Otherwise, search the right side, using the node on the right as the new root
else if (number > tree->number)
{
    return search(tree->right, number);
}
// Finally, we've found the number we're looking for, so we can return true.
// We can simplify this to just "else", since there's no other case possible
else if (number == tree->number)
{
    return true;
}
}
```

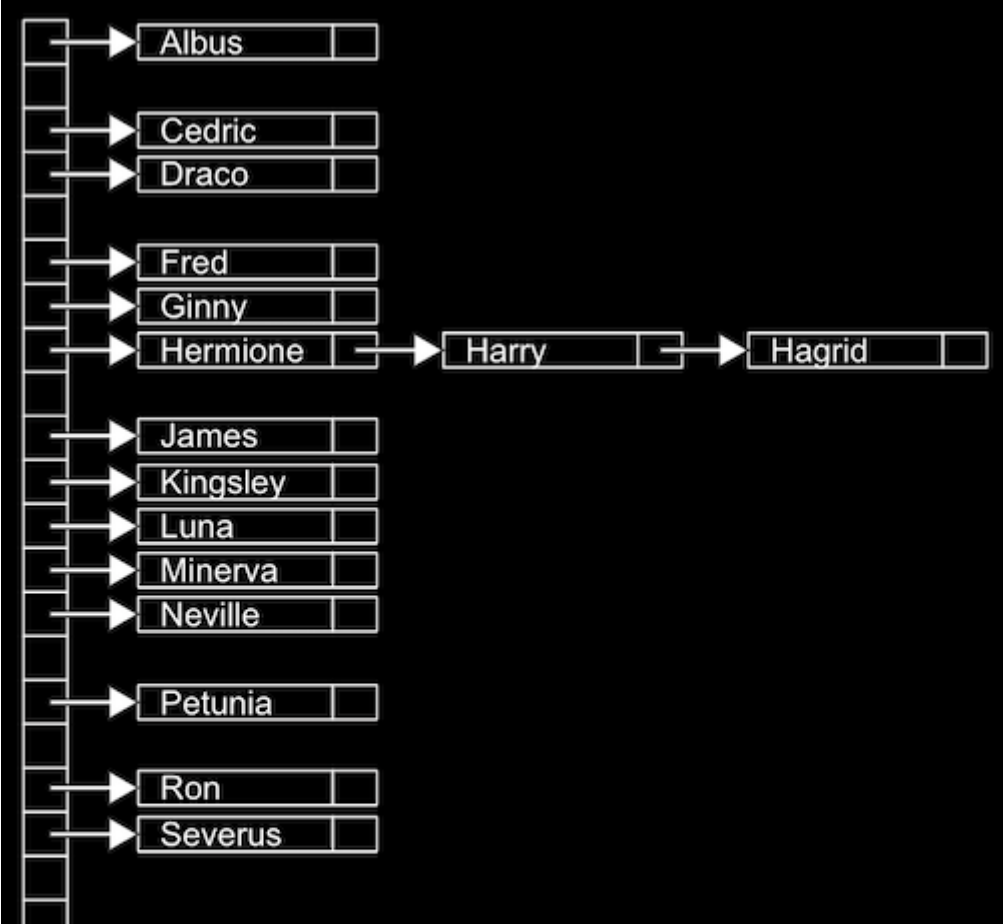
- Con un árbol de búsqueda binario, hemos incurrido en el costo de incluso más memoria, ya que cada nodo ahora necesita espacio para un valor y dos punteros. Insertar un nuevo valor tomaría $O(\log n)$ tiempo, ya que necesitamos encontrar los nodos entre los que debería ir.
- Sin embargo, si agregamos suficientes nodos, nuestro árbol de búsqueda podría comenzar a verse como una lista vinculada:



- Comenzamos nuestro árbol con un nodo con valor de 1, luego agregamos el nodo con valor 2 y finalmente agregamos el nodo con valor 3. Aunque este árbol sigue las restricciones de un árbol de búsqueda binario, no es tan eficiente como podría ser.
- Podemos hacer que el árbol esté equilibrado, u óptimo, haciendo que el nodo con valor sea 2 el nuevo nodo raíz. Los cursos más avanzados cubrirán estructuras de datos y algoritmos que nos ayudan a mantener los árboles equilibrados a medida que se agregan nodos.

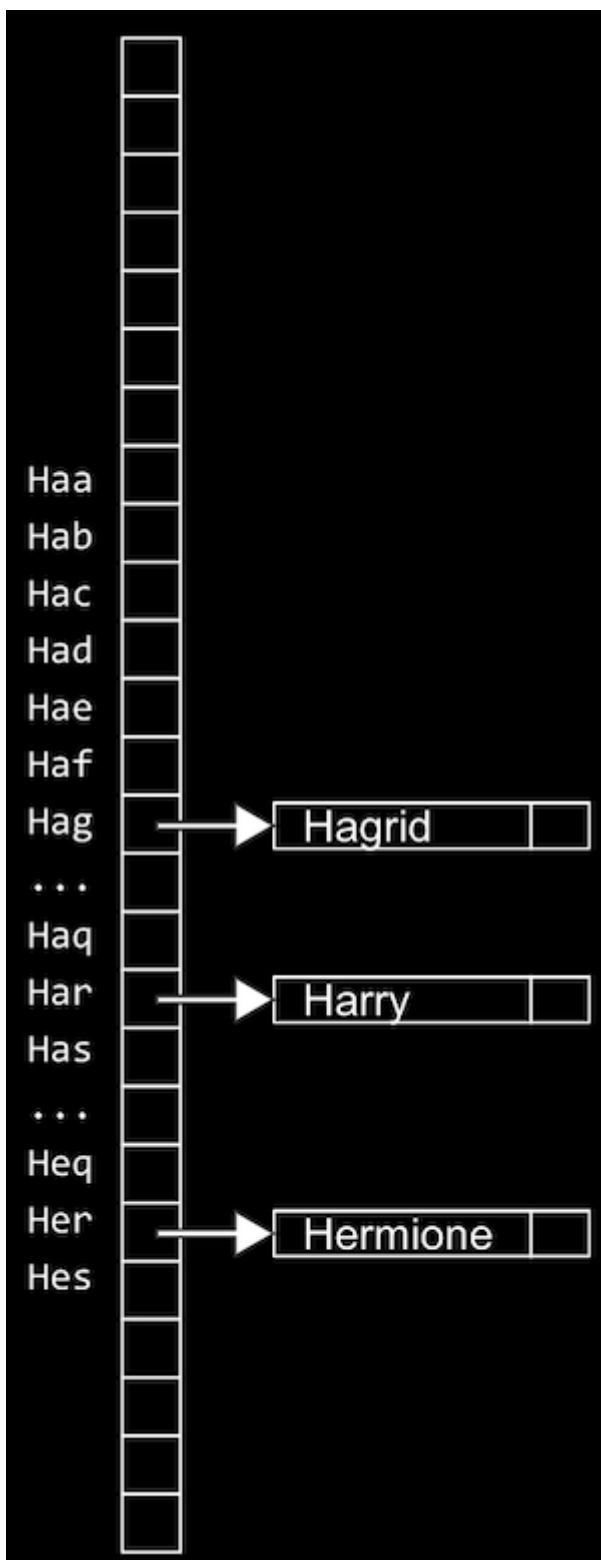
Más estructuras de datos

- Una estructura de datos con un tiempo casi constante, la búsqueda $O(1)$ es una **tabla hash**, que es esencial una matriz de listas enlazadas. Cada lista vinculada en la matriz tiene elementos de una determinada categoría.
- Por ejemplo, podríamos tener muchos nombres y podríamos ordenarlos en una matriz con 26 posiciones, una para cada letra del alfabeto:

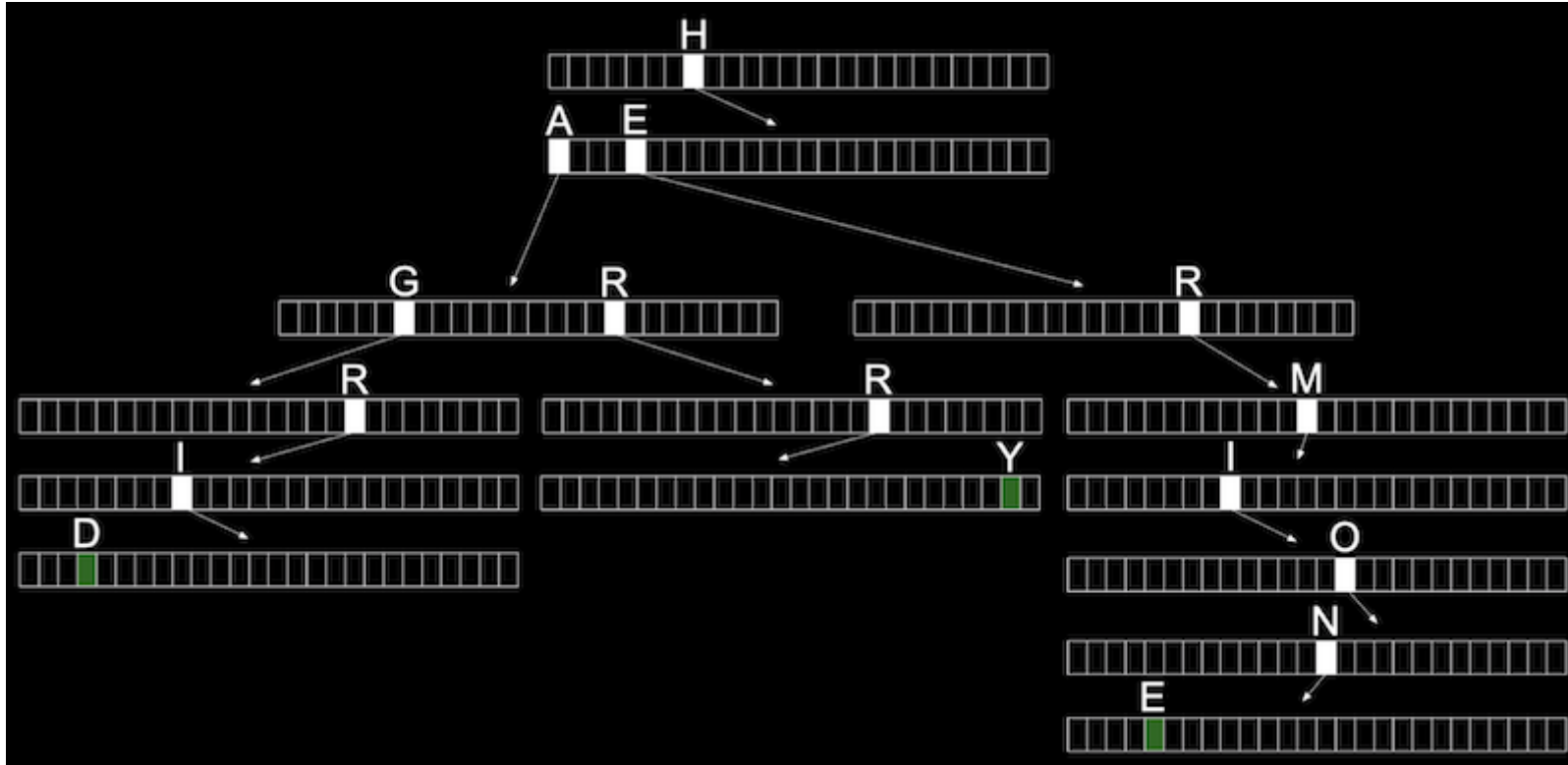




- Dado que tenemos acceso aleatorio con matrices, podemos establecer elementos e indexar en una ubicación, o depósito, en la matriz rápidamente.
- Una ubicación puede tener varios valores coincidentes, pero podemos agregar un valor a otro valor, ya que son nodos en una lista vinculada, como vemos con Hermione, Harry y Hagrid. No necesitamos aumentar el tamaño de nuestra matriz ni mover ninguno de nuestros otros valores.
- Esto se llama tabla hash porque usamos una **función hash**, que toma alguna entrada y la asigna de manera determinista a la ubicación en la que debería ir. En nuestro ejemplo, la función hash solo devuelve un índice correspondiente a la primera letra del nombre, como en cuanto `0` a "Albus" y `25` "Zacharias".
- Pero en el peor de los casos, todos los nombres pueden comenzar con la misma letra, por lo que podríamos terminar con el equivalente a una sola lista vinculada nuevamente. Podríamos mirar las dos primeras letras y asignar suficientes depósitos para $26 * 26$ posibles valores hash, o incluso las primeras tres letras, lo que requiere $26 * 26 * 26$ depósitos:



- Ahora, estamos usando más espacio en la memoria, ya que algunos de esos depósitos estarán vacíos, pero es más probable que solo necesitemos un paso para buscar un valor, lo que reduce nuestro tiempo de ejecución para la búsqueda.
- Para clasificar algunas cartas de juego estándar, también, primero podríamos comenzar colocándolas en montones por palo, de espadas, diamantes, corazones y tréboles. Luego, podemos clasificar cada pila un poco más rápido.
- Resulta que el peor tiempo de ejecución para una tabla hash es $O(n)$, ya que, a medida que n se vuelve muy grande, cada depósito tendrá valores del orden de n , incluso si tenemos cientos o miles de depósitos. Sin embargo, en la práctica, nuestro tiempo de ejecución será más rápido ya que dividimos nuestros valores en varios depósitos.
- En el conjunto de problemas 5, seremos desafiados a mejorar el tiempo de ejecución en el mundo real de la búsqueda de valores en nuestras estructuras de datos, al mismo tiempo que equilibramos nuestro uso de la memoria.
- Podemos usar otra estructura de datos llamada **trie** (que se pronuncia como "intentar" y es la abreviatura de "recuperación"). Un trie es un árbol con matrices como nodos:



- Cada matriz tendrá almacenada cada letra, AZ. Para cada palabra, la primera letra apuntará a una matriz, donde la siguiente letra válida apuntará a otra matriz, y así sucesivamente, hasta que alcancemos un valor booleano que indique el final de una palabra válida, marcada en verde arriba. Si nuestra palabra no está en el trie, entonces una de las matrices no tendrá un puntero o carácter de terminación para nuestra palabra.
- En el trie anterior, tenemos las palabras Hagrid, Harry y Hermione.
- Ahora, incluso si nuestra estructura de datos tiene muchas palabras, el tiempo máximo de búsqueda será solo la longitud de la palabra que estamos buscando. Este podría ser un máximo fijo, por lo que podemos tener $O(1)$ para la búsqueda y la inserción.
- Sin embargo, el costo de esto es que necesitamos mucha memoria para almacenar punteros y valores booleanos como indicadores de palabras válidas, aunque muchas de ellas no se usarán.
- Incluso hay construcciones de nivel superior, **estructuras de datos abstractas**, donde usamos nuestros bloques de construcción de matrices, listas vinculadas, tablas hash e intentamos *implementar* una solución a algún problema.
- Por ejemplo, una estructura de datos abstracta es una **cola**, como una fila de personas esperando, donde el primer valor que ingresamos son los primeros valores que se eliminan, o primero en entrar, primero en salir (FIFO). Para agregar un valor, lo ponemos en **cola**, y para eliminar un valor, lo **quitamos de la cola**. Esta estructura de datos es abstracta porque es una idea que podemos implementar de diferentes maneras: con una matriz que cambiamos de tamaño a medida que agregamos y eliminamos elementos, o con una lista vinculada donde agregamos valores al final.
- Una estructura de datos "opuesta" sería una **pila**, donde los elementos agregados más recientemente se eliminan primero: último en entrar, primero en salir (LIFO). En una tienda de ropa, podríamos tener, o **pop**, el jersey encima de una pila, y se añadiría nuevos suéteres, o **empujado**, a la parte superior también.
- Otro ejemplo de una estructura de datos abstracta es un **diccionario**, donde podemos asignar claves a valores, como palabras a sus definiciones. Podemos implementar uno con una tabla hash o una matriz, teniendo en cuenta la compensación entre tiempo y espacio.
- Echamos un vistazo a "[Jack aprende los hechos sobre colas y pilas](https://www.youtube.com/watch?v=ItAG3s6KIEI)" (<https://www.youtube.com/watch?v=ItAG3s6KIEI>), una animación sobre estas estructuras de datos.