CS50 - SECTION 4

10/9/18

POST-MORTEM ON PSET 3

REMINDERS

1. Read the problem set specification and make sure you *really* understand what is going on
   - What's the purpose of `RGBTRIPLE` and what does it represent?
   - Why is there padding in a bitmap file?
   - What are the different functions in `copy.c` and how does that file work?

REMINDERS

2. If the walkthrough offers you two options, think through why one may be better than another.
    - What are the advantages/disadvantages of the scanline array approach vs pixel-by-pixel for reading in the input file?

REMINDERS

3. Be *proactive* with your debugging.
    - e.g. **`peek`**, **`diff`**, etc. for this pset specifically
    - **`help50`**, **`debug50`**, and **`valgrind`** as a whole
        - **`valgrind`** for segmentation faults especially

REMINDERS

3. Be *proactive* with your debugging.
    - e.g. **`peek`**, **`diff`**, etc. for this pset specifically
    - **`help50`**, **`debug50`**, and **`valgrind`** as a whole
        - **`valgrind`** for segmentation faults especially

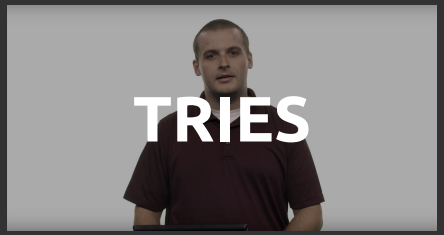
CONCEPTS DEEP-DIVE

“SHORTS” FOR THE WEEK

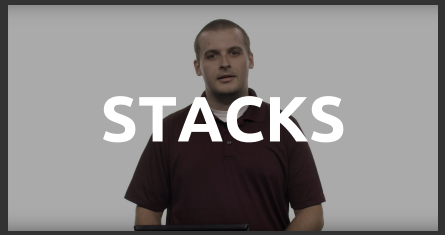
https://youtu.be/crxfzK3Oc9M

https://youtu.be/xdkSNe43iNM

https://youtu.be/a97eCq6EN88

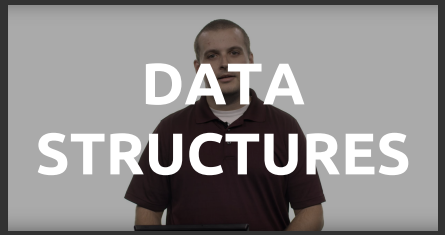


https://youtu.be/MTxh0kx1Vvs



https://youtu.be/2JVse9x1rug

https://youtu.be/XVezfHlhZjk



https://youtu.be/Ryz5KK5G8Sc

STRUCTURES IN C

Thus far, we've seen data types which are singular and narrow in their purpose:

- `int`
- `char *`
- `long`
- `float`
- etc.

STRUCTURES IN C

But what if we need to represent more complex data structures in memory that are not so singular?

STRUCTURES IN C

But what if we need to represent more complex data structures in memory that are not so singular?

We can use utilize structures in C to help create new data types composed of variables of any number of different types themselves

STRUCTURES IN C

```
struct car
{
    int year;
    char model[10];
    char plate[7];
    bool automatic;
};
```

STRUCTURES IN C

```
struct car
{
    int year;
    char model[10];
    char plate[7];
    bool automatic;
};
```

This `car` structure contains four different “members” for which we can assign values to when we assign a variable to it

STRUCTURES IN C

Note that you can define your structures at the top of your C file or in another header file (.h)

When might you use one approach over another?

STRUCTURES IN C

Note that you can define your structures at the top of your C file or in another header file (.h)

When might you use one approach over another?

We generally use a header file if we want multiple programs to access our structure or just put it in the main file if there isn't high reusability.

STRUCTURES IN C

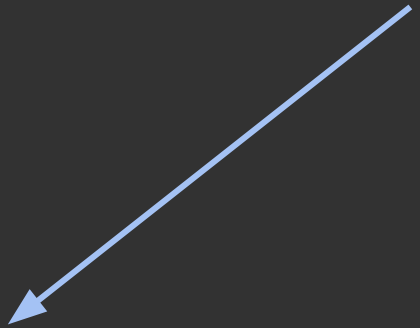
You can access the members/fields of your structure using the dot operator:

```
// declaration
struct car mycar;

// field access
mycar.year = 2011;
strcpy(mycar.plate, "CS50");
mycar.automatic = false;
```

STRUCTURES IN C

You can access the members/fields of your structure using the dot operator:

Why use `strcpy()` here?

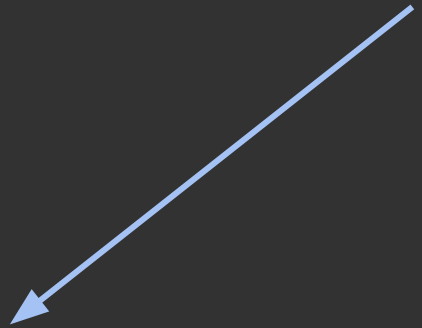


```
// declaration
struct car mycar;

// field access
mycar.year = 2011;
strcpy(mycar.plate, "CS50");
mycar.automatic = false;
```

STRUCTURES IN C

You can access the members/fields of your structure using the dot operator:

```
// declaration
struct car mycar;

// field access
mycar.year = 2011;
strcpy(mycar.plate, "CS50");
mycar.automatic = false;
```



Why use `strcpy()` here?

Because `mycar.plate` is a character array and thus we can't just set two arrays equal to one another

STRUCTURES IN C

We can also create new structures using dynamically allocated memory like any other data type in C:

```
struct car *mycar = malloc(sizeof(struct car));
```

STRUCTURES IN C

To access the data inside a dynamically allocated struct, we need to first deference the pointer and then access the field we want:

```
struct car *mycar = malloc(sizeof(struct car));
int y = (*mycar).year;
```

STRUCTURES IN C

We have a shortcut for this process in C too:

```
struct car *mycar = malloc(sizeof(struct car));
int y = (*mycar).year;


int x = mycar->year;
```

STRUCTURES IN C

Instead of having to retype "`struct car`" every time, we can rename the structure using `typedef`:

```
typedef old-name new-name;
```

STRUCTURES IN C

Instead of having to retype "`struct car`" every time, we can rename the structure using `typedef`:

```
typedef old-name new-name;
```

```
typedef char * string;
```

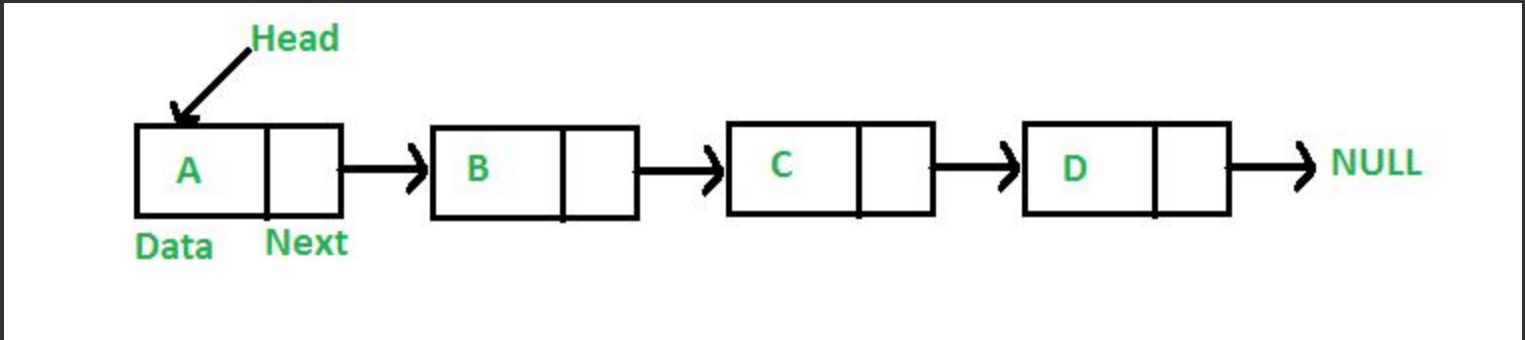
STRUCTURES IN C

Instead of having to retype "`struct car`" every time, we can rename the structure using `typedef`:

```
typedef old-name new-name;
typedef char * string;
typedef struct car car;
```

LINKED LISTS

A **linked list** is a data structure in C that allows us to create a chain of nodes that is dynamically-sized:



LINKED LISTS

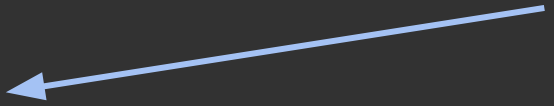
The fundamental basis of any linked list is the `node` structure:

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

LINKED LISTS

The fundamental basis of any linked list is the `node` structure:

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



This can be any value you want. You could have a linked list of strings, integers, floats, etc. Just adjust the type as needed

LINKED LISTS

What are the advantages of a linked list?

LINKED LISTS

What are the advantages of a linked list?

- **They’re dynamically-sized, so we can add/remove elements (arrays are fixed!)**
- **As a corollary, insertion/deletion is really easy**
- **It efficiently uses memory: we just unlink nodes when we no longer need them**

LINKED LISTS

What are the disadvantages of a linked list?

LINKED LISTS

What are the disadvantages of a linked list?

- **They take memory memory to store: an array just stores the elements, but linked lists need the node structure**
- **You have to traverse multiple elements to access a single node (you can’t just go directly to a specific index like you can with arrays)**

LINKED LISTS

Five operations to know how to do on a linked list:

1. Creating a linked list when it doesn't exist.
2. Searching through a linked list to find an element.
3. Inserting a new node into a linked list.
4. Deleting an entire linked list.
5. Deleting a single element from a linked list.

#5 isn't on the pset, but is valuable to know

HANDS ON PRACTICE

http://bit.ly/2CBsVWM

LINKED LISTS

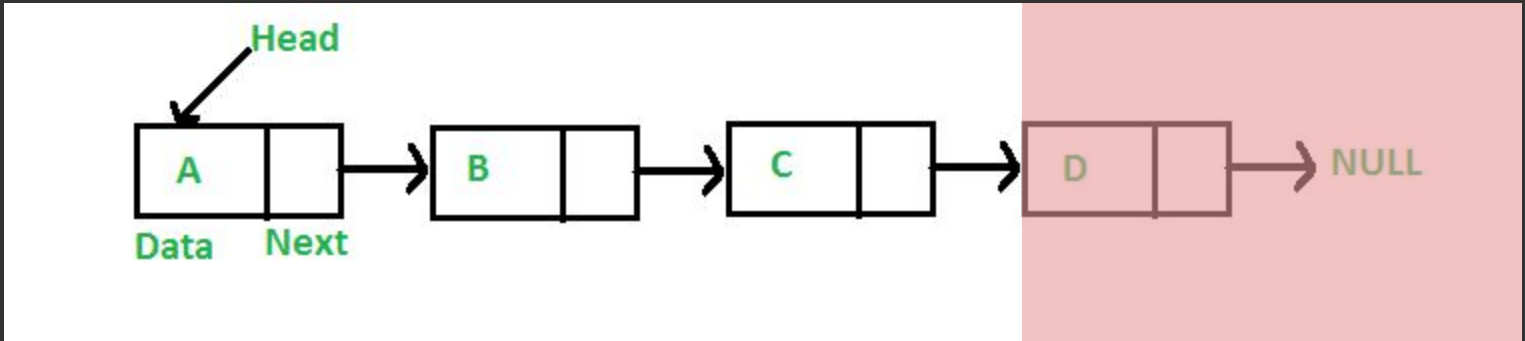
1. Creating a linked list when it doesn’t exist.

We need to:

- Dynamically allocate space for a new node.
- Check to make sure we didn’t run out of memory.
- Initialize the value field.
- Initialize the next field (specifically, to NULL).
- Return a pointer to your newly created node.

LINKED LISTS

1. Creating a linked list when it doesn't exist.



LINKED LISTS

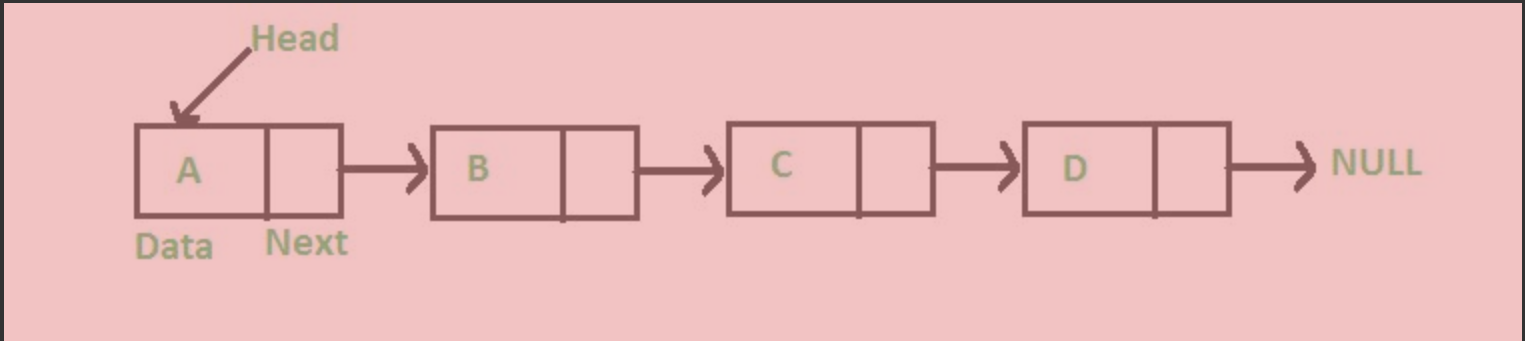
2. Searching through a linked list to find an element.

We need to:

- Create a traversal pointer pointing to the list's head (first element).
- If the current node's value field is what we're looking for, return true. If not, set the traversal pointer to the next pointer in the list and go back to the previous step.
- If you've reached the end of the list, return whether the last element is the one we want.

LINKED LISTS

2. Searching through a linked list to find an element.



LINKED LISTS

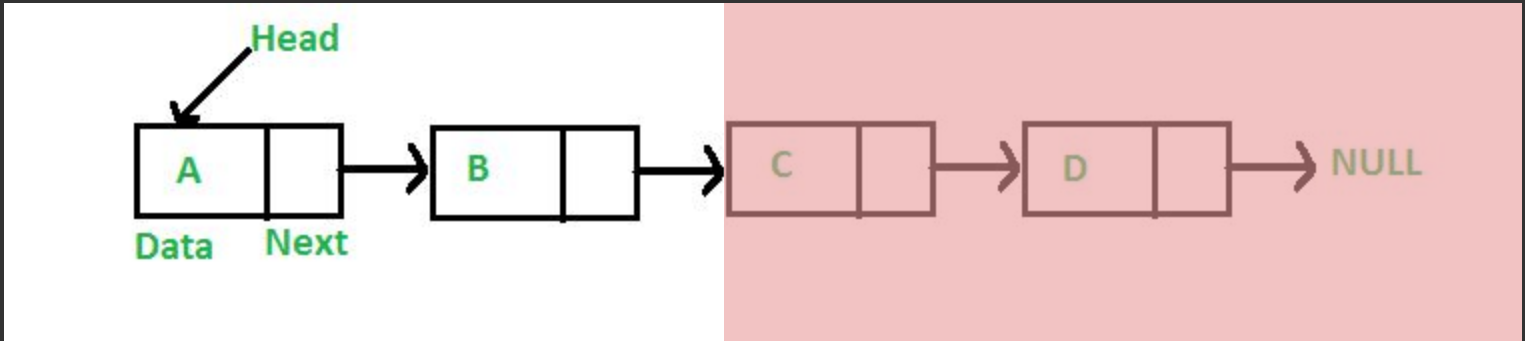
3. Inserting a new node into a linked list.

We need to:

- Dynamically allocate space for a new linked list node.
- Check to make sure we didn’t run out of memory.
- Populate and insert the node at the beginning of the linked list.
- Return a pointer to the new head of the linked list.

LINKED LISTS

3. Inserting a new node into a linked list.



LINKED LISTS

4. Deleting an entire linked list.

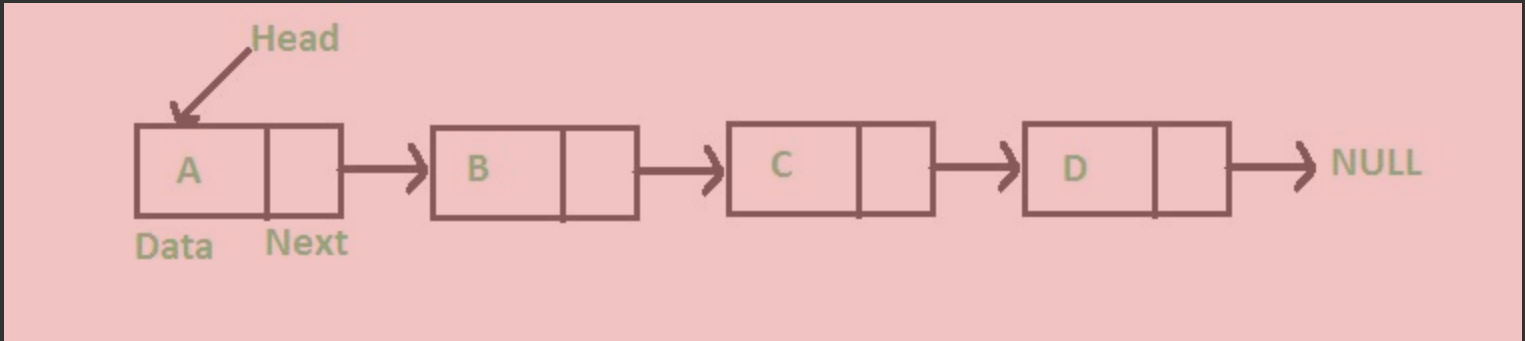
We need to:

- If you've reached a `NULL` pointer, stop.
- Delete the rest of the list.
- Free the current node.

This would require a recursive solution!

LINKED LISTS

4. Deleting an entire linked list.



HANDS ON PRACTICE - SOLUTIONS

http://bit.ly/2C3kUZP

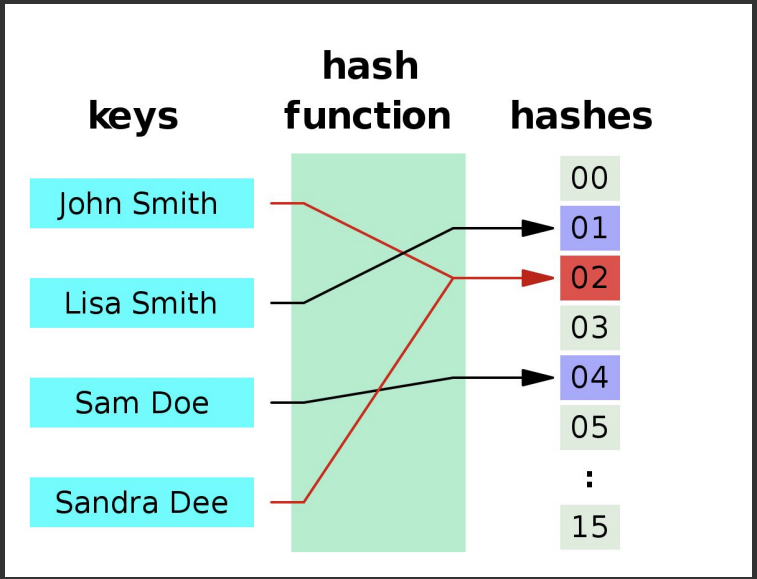
HASH TABLES

What is a **hash function**?

HASH TABLES

What is a **hash function**?

“A hash function is any function that can be used to map data of arbitrary size to data of a fixed size.”



HASH TABLES

A **hash table** is a data structure which uses a hash function and an array to determine where in the array to store elements

You run your elements through the hash function, get back a hash value, and then use that as index in your array to store the value

HASH TABLES

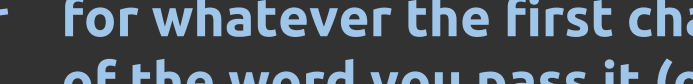
A really simple example of this:

```c
#include <string.h>

int hash(char *word);

int main(void) {
    // Creating an array for the hash table
    char *hashTable[26];

    // Adding an element to the hash table
    char *firstWord = "Hello";
    strcpy(hashTable[hash(firstWord)], firstWord);
}

// Hash Function
int hash(char *word) {
    return  (int) word[0] - 'A';
}
```

HASH TABLES

A really simple example of this:

```c
#include <string.h>

int hash(char *word);

int main(void) {
    // Creating an array for the hash table
    char *hashTable[26];

    // Adding an element to the hash table
    char *firstWord = "Hello";
    strcpy(hashTable[hash(firstWord)], firstWord);
}

// Hash Function
int hash(char *word) {
    return  (int) word[0] - 'A';
}
```



The hash function literally just returns the “alphabetic index” for whatever the first character of the word you pass it (only works for capital letters)

HASH TABLES

Why might this be a bad hash table?

```c
#include <string.h>

int hash(char *word);

int main(void) {
    // Creating an array for the hash table
    char *hashTable[26];

    // Adding an element to the hash table
    char *firstWord = "Hello";
    strcpy(hashTable[hash(firstWord)], firstWord);
}

// Hash Function
int hash(char *word) {
    return  (int) word[0] - 'A';
}
```

HASH TABLES

Why might this be a bad hash table?

```c
#include <string.h>

int hash(char *word);

int main(void) {
    // Creating an array for the hash table
    char *hashTable[26];

    // Adding an element to the hash table
    char *firstWord = "Hello";
    strcpy(hashTable[hash(firstWord)], firstWord);
}

// Hash Function
int hash(char *word) {
    return  (int) word[0] - 'A';
}
```

The array it uses is small! Words will quickly fill up every slot and the slots are uneven (‘Z’ will be referenced a lot less than ‘T’).

HASH TABLES

What makes for a good hash function?

HASH TABLES

What makes for a good hash function?

- **Use only the data being hashed.**
- **Use all of the data being hashed.**
- **Be deterministic (same result every time given same input; no randomness!).**
- **Uniformly distribute data.**
- **Generate very different hash codes for very similar data.**

HASH TABLES

What do we do when we get a collision?

HASH TABLES

What do we do when we get a collision?

We can solve it in two ways:

1. Linear probing
2. Chaining

HASH TABLES

Linear Probing:

- If we have a collision, try to place the data instead in the next consecutive element of the array, trying each consecutive element until we find a vacancy.
- That way, if we don't find it right where we expect it, it's hopefully nearby.

Can lead to **clustering**—statistically more likely to form "clusters" with other collisions reducing effectiveness of the hash table; Also, # of elements is capped at the size of our array

HASH TABLES

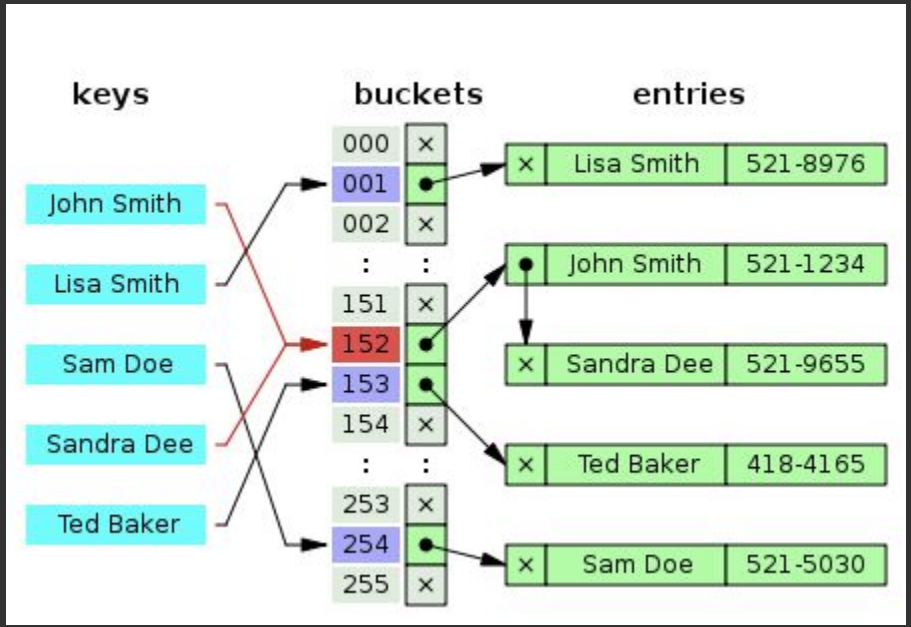
Chaining:

- Each element of the array is now a pointer to the head of a linked list.
- If we have a collision, create a new node and add it to the chain at that location.
- That way, if it's in the chain at the hash code location, it's in the data structure.

Not subject to the clustering problem from before and our array can now store as many elements as you have memory for

HASH TABLES

Chaining:



KEY-VALUE PAIRS

Many of the data structures we've discussed in CS50 thus far conform to the *key-value pair* pattern:

You have a **key** which is unique or mostly unique for each **value** that it represents

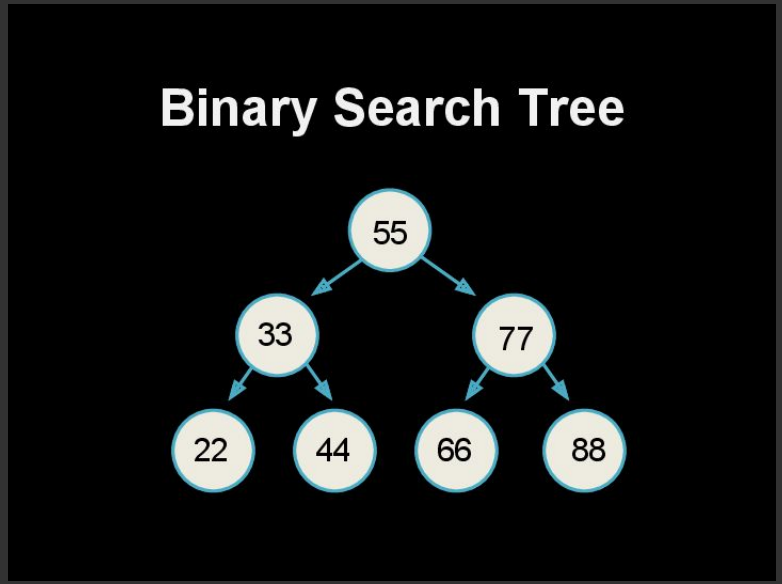
KEY-VALUE PAIRS

Examples:

- Arrays - The index is the key and the data at that location is the value
- Hash Tables - The hash code of the data is the key and the slot in the array at that index is the value

TREES

- A **tree** is a hierarchical data type in computer science with *many* applications



TRIES

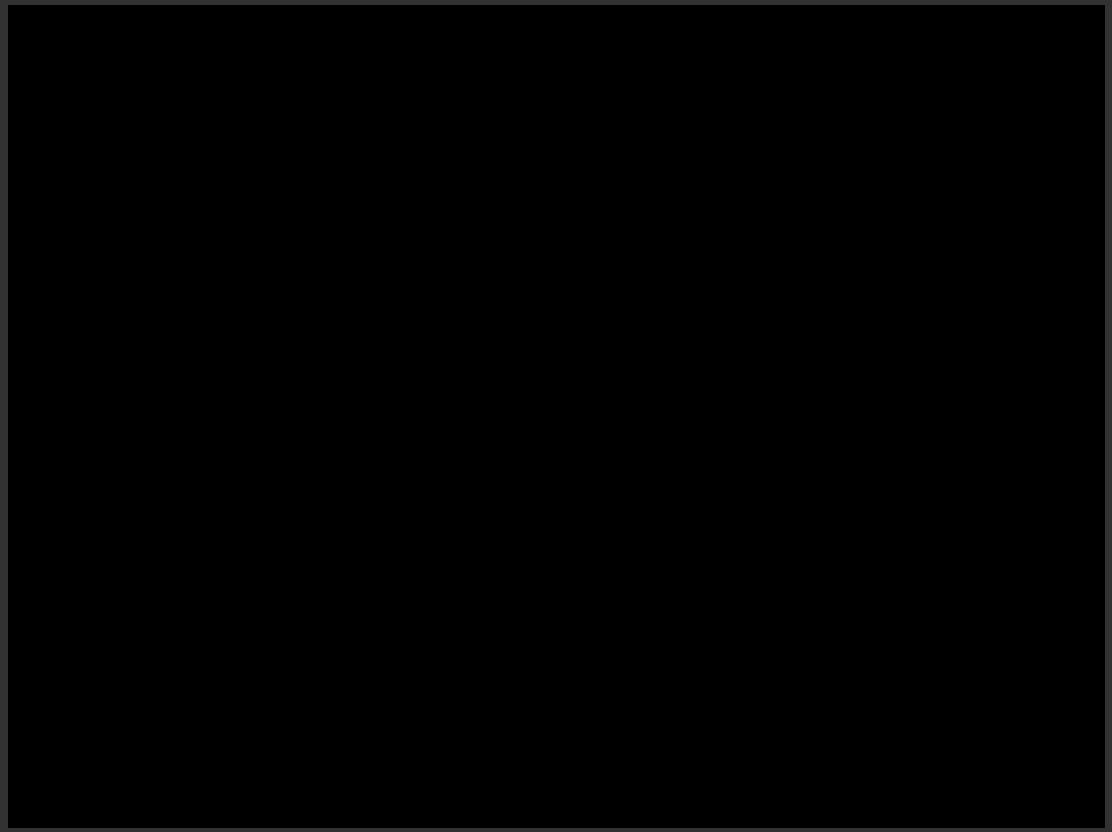
- One type of tree is a **trie**, an ordered tree optimized for searching (usually strings)
- Tries follow the key-value pattern too, but *implicitly*
  - There is no explicit key defined for each element; Instead, the position of the element in the tree is its key
  - The data located at that position is the value

TRIES

A simple trie to store years (keys) and the universities founded during those years (values):

```
typedef struct trie
{
    char university[20];
    struct trie *paths[10];
}
node;
```

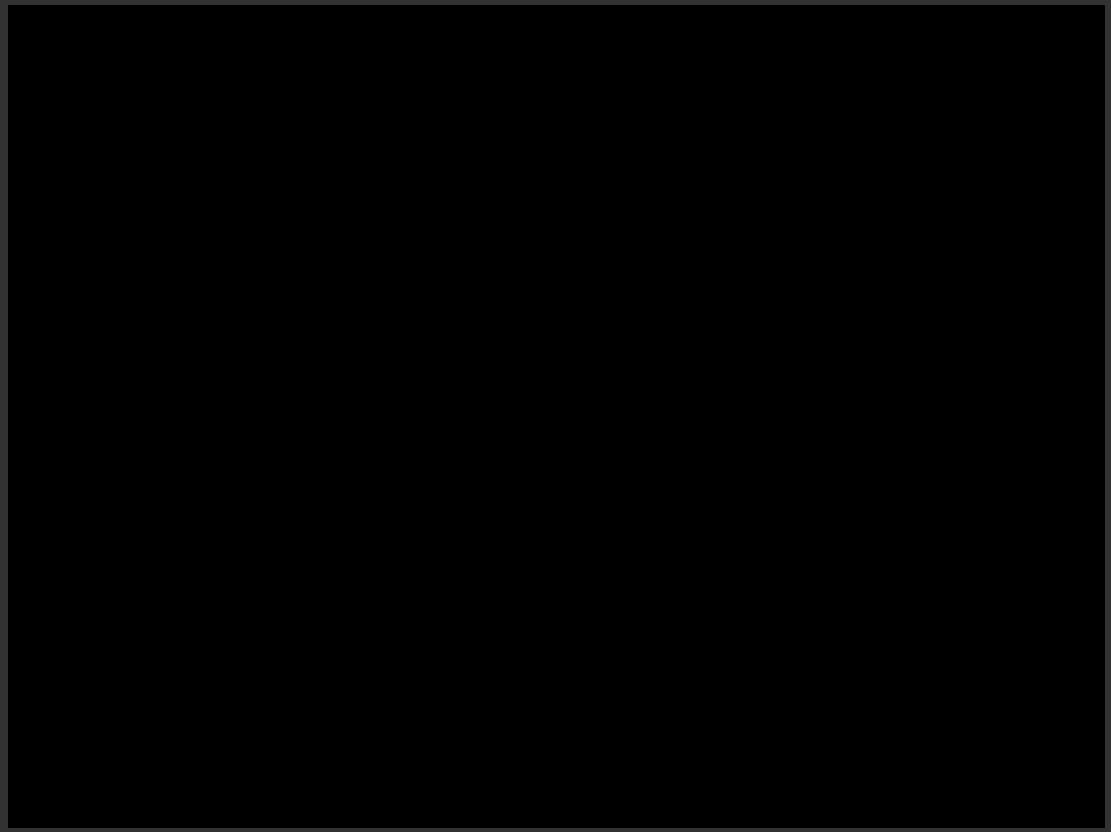
TRIES



TRIES

To search for an element in the trie, use successive digits to navigate from the root, and if you make it to the end without hitting a dead end (a `NULL` pointer in this case), simply read the data at your current location.

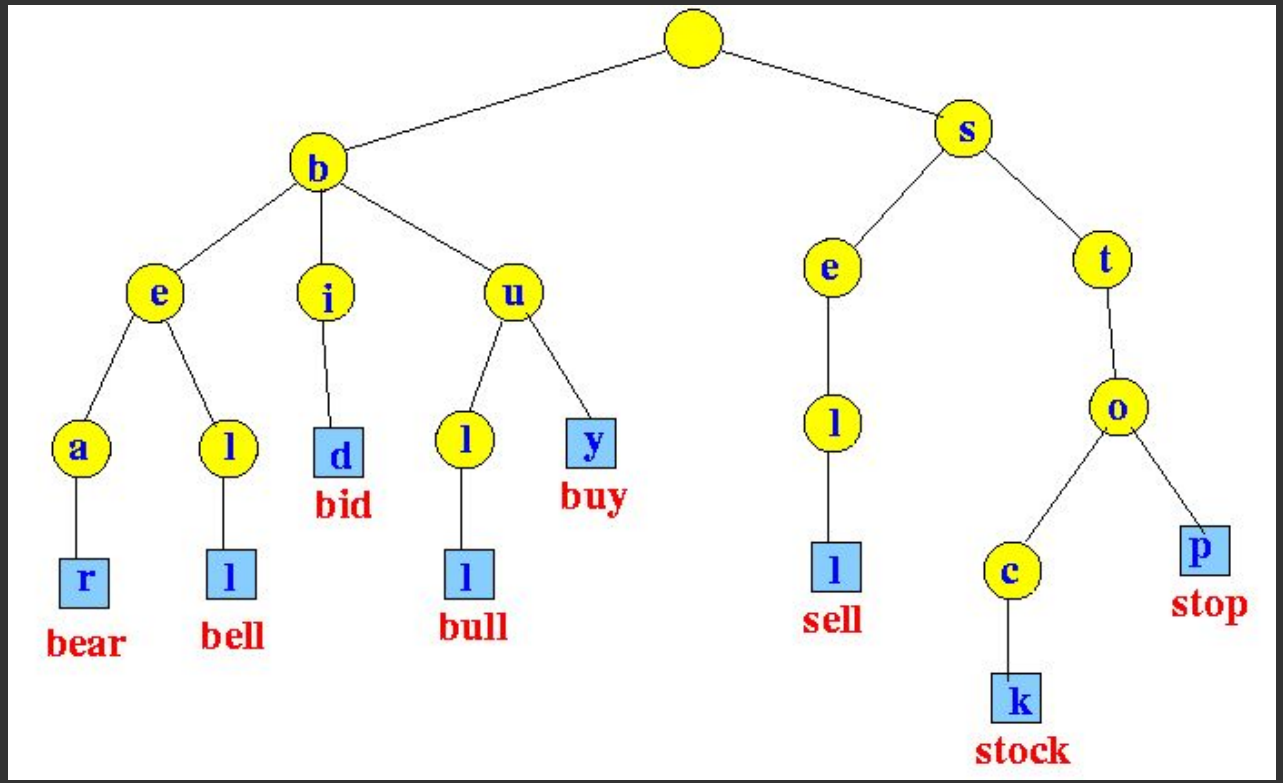
TRIES



TRIES

It’s more common to represent words in a trie as the values (guaranteed to be unique as opposed to the example from before—two universities could have been founded in the same year)

TRIES


b
s
e
i
u
e
t
a
l
d
l
y
l
o
r
l
l
l
c
p
bear
bell
bid
bull
buy
sell
stop
k
stock

TRIES

What are the advantages and disadvantages of a trie?

TRIES

What are the advantages and disadvantages of a trie?

- They are very fast because they have a constant lookup time (based off of the size of the word)
- If constructed off unique keys, collisions are impossible
- They need a lot of memory—Each node must point to all different possibilities at that node (e.g. each letter of the alphabet at *every level*)
- A very good hash table might have even faster lookup depending on how you write your hash function

STACKS AND QUEUES

No time to discuss these in section, but check out these resources:

- https://www.geeksforgeeks.org/stack-data-structure-introduction-program/
- https://www.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/
- https://www.hackerearth.com/practice/notes/stacks-and-queues/

PROBLEM SET 4 PREVIEW

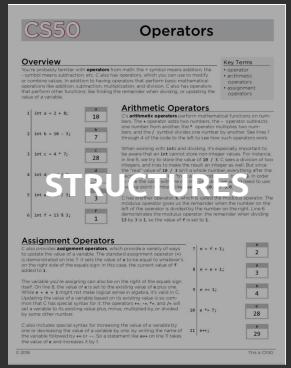
PROBLEM SET 4 PREVIEW

Implement `speller.c` using ONE of the following:

- Hash tables
- Tries

Hash tables are probably the best bet if you're feeling less comfortable.

REFERENCE SHEETS



https://www.dropbox.com/sh/5y662ey1hc4sde4/AACjqHN3NtSKk4ShsRDFd_Sja?dl=0&m=&preview=Structures+and+Encapsulation.pdf

FINAL NOTES

- ★ Check out CS50 Tutoring if you need some one-on-one help with the concepts—it's free!
- ★ Remember what we discussed from the post-mortem today: Ensure you fully understand the concepts and what you're trying to before diving into code
  - ○ When in doubt, start on paper and *then* move to the computer!