This is CS50
data structures
abstract data types
queues
FIFO
enqueue

decqueue
stacks
LIFO
push

pop
typedef struct
{
    person people[CAPACITY];
    int size;
}
stack;
const int CAPACITY = 50;

typedef struct
{
    person people[CAPACITY];
    int size;
}
stack;
This is CS50
arrays
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>hell</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>,</td>
<td>w</td>
<td>o</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\theta</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

World, hello, 123
data structures
struct

*
struct ->
linked lists
typedef struct
{
    string name;
    string number;
}
person;
typedef struct
{
    char *name;
    char *number;
}
person;
typedef struct
{
}

person;
typedef struct
{
  
}
node;
typedef struct {
    int number;
} node;
typedef struct
{
    int number;
    node *next;
}
node;
typedef struct node
{
    int number;
    node *next;
}
node;
typedef struct node
{
    int number;
    struct node *next;
}
node;
This is CS50
node *list;
node *list;
node *list = NULL;
node *list = NULL;
node *n = malloc(sizeof(node));
node *n = malloc(sizeof(node));
node *n = malloc(sizeof(node));
node *n = malloc(sizeof(node));
(*n).number = 1;
n->number = 1;
n->next = NULL;
n->next = NULL;
list = n;
list = n;
list

1
node *n = malloc(sizeof(node));
node *n = malloc(sizeof(node));
node *n = malloc(sizeof(node));
node *n = malloc(sizeof(node));
n->number = 2;
n->number = 2;
n->next = NULL;
n->next = NULL;
list = n;
list = n;
list = n;
list

2

1

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

2

1
list

ptr

1

2

3
list

1

2

3
$O(n^2)$

$O(n \log n)$

$O(n)$

$O(n)$

$O(\log n)$

$O(\log n)$

$O(1)$
list
list
list -> 1
\( O(n^2) \)
\( O(n \log n) \)
\( O(n) \)
\( O(\log n) \)
\( O(1) \)
list

1

2
list

1

2

3

4
$O(n^2)$

$O(n \log n)$

$O(n)$

$O(\log n)$

$O(1)$
trees
binary search trees
typedef struct node
{
    int number;
    struct node *next;
}
node;
typedef struct node
{
    int number;
}
node;
typedef struct node
{
    int number;
}
node;
typedef struct node
{
    int number;
    struct node *left;
    struct node *right;
}
node;
bool search(node *tree, int number)
{
    if (tree == NULL)
    {
        return false;
    }
    else if (number < tree->number)
    {
        return search(tree->left, number);
    }
    else if (number > tree->number)
    {
        return search(tree->right, number);
    }
    else if (number == tree->number)
    {
        return true;
    }
}
bool search(node *tree, int number)
{
    if (tree == NULL)
    {
        return false;
    }
    else if (number < tree->number)
    {
        return search(tree->left, number);
    }
    else if (number > tree->number)
    {
        return search(tree->right, number);
    }
    else if (number == tree->number)
    {
        return true;
    }
}
bool search(node *tree, int number)
{
    if (tree == NULL)
    {
        return false;
    }
    else if (number < tree->number)
    {
        return search(tree->left, number);
    }
    else if (number > tree->number)
    {
        return search(tree->right, number);
    }
    else if (number == tree->number)
    {
        return true;
    }
}
bool search(node *tree, int number)
{
    if (tree == NULL)
    {
        return false;
    }
    else if (number < tree->number)
    {
        return search(tree->left, number);
    }
    else if (number > tree->number)
    {
        return search(tree->right, number);
    }
    else if (number == tree->number)
    {
        return true;
    }
}
bool search(node *tree, int number)
{
    if (tree == NULL)
    {
        return false;
    }
    else if (number < tree->number)
    {
        return search(tree->left, number);
    }
    else if (number > tree->number)
    {
        return search(tree->right, number);
    }
    else if (number == tree->number)
    {
        return true;
    }
}
bool search(node *tree, int number)
{
    if (tree == NULL)
    {
        return false;
    }
    else if (number < tree->number)
    {
        return search(tree->left, number);
    }
    else if (number > tree->number)
    {
        return search(tree->right, number);
    }
    else
    {
        return true;
    }
}
\(O(n^2)\)
\(O(n \log n)\)
\(O(n)\)
\(O(n)\)
\(O(\log n)\)
\(O(\log n)\)
\(O(1)\)
dictionaries
John Harvard

+1 (949) 468-2750
The diagram illustrates the relationship between the size of the problem and the time to solve it. The green curve represents $O(n)$, indicating linear time complexity, while the red curve represents $O(\log n)$, indicating logarithmic time complexity. As the size of the problem increases, the time to solve with $O(n)$ grows linearly, whereas with $O(\log n)$, the growth is much slower, reflecting the efficiency of algorithms with logarithmic complexity.
The graph illustrates the time to solve a problem in terms of the size of the problem. The time complexities are shown for:

- $O(n)$, represented by the red line,
- $O(\log n)$, represented by the yellow line,
- $O(1)$, represented by the green dashed line.

As the size of the problem increases, the $O(n)$ line shows linear growth, the $O(\log n)$ line shows logarithmic growth, and the $O(1)$ line represents a constant time complexity, indicating that the time to solve the problem does not change with the size of the problem.
hashing
hash function
hash tables
<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albus</td>
</tr>
<tr>
<td>Cedric</td>
</tr>
<tr>
<td>Draco</td>
</tr>
<tr>
<td>Fred</td>
</tr>
<tr>
<td>Ginny</td>
</tr>
<tr>
<td>Hermione</td>
</tr>
<tr>
<td>James</td>
</tr>
<tr>
<td>Kingsley</td>
</tr>
<tr>
<td>Luna</td>
</tr>
<tr>
<td>Minerva</td>
</tr>
<tr>
<td>Neville</td>
</tr>
<tr>
<td>Petunia</td>
</tr>
<tr>
<td>Ron</td>
</tr>
<tr>
<td>Severus</td>
</tr>
<tr>
<td>Vernon</td>
</tr>
<tr>
<td>Zacharias</td>
</tr>
</tbody>
</table>
Hermione
Hermione
Harry
Hermione
Haa
Hab
Hac
Had
Hae
Haf
Hag
...  
Haq
Har
Has
...  
Heq
Her
Hes
Hermione
\( O(n^2) \)
\( O(n \log n) \)
\( O(n) \)
\( O(\log n) \)
\( O(1) \)
typedef struct
{
    char *name;
    char *number;
}
person;
typedef struct node
{
    char *name;
    char *number;
    struct node *next;
}
node;
node *table[26];
hash function
Zacharias $\rightarrow$ 25
$O(n)$
\(O(n/k)\)
$O(n)$
tries
typedef struct node
{
    char *number;
    struct node *children[26];
}
node;
node *trie;
$O(n^2)$

$O(n \log n)$

$O(n)$

$O(\log n)$

$O(1)$
This is CS50