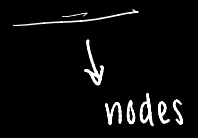
Watch this video! We'll come back to hash tables at the end of the class.

tinyurl.com/phyllis-hashtable

CS50 Section 5

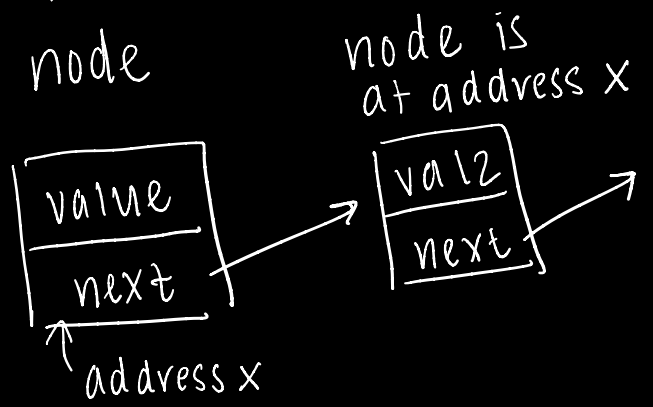
Linked Lists

- We refer to this combination of structs and pointers, when used to create a “chain” of nodes a **linked list.**



- A linked list **node** is a special type of struct with two fields:
  - Data of some type
  - A pointer to another linked list node.

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



- Create a linked list:
  - Dynamically allocate space for a new (your first!) node. malloc!
  - Check to make sure you didn't run out of memory (does node == NULL?)
  - Initialize the value field.
  - Initialize the next field (specifically, to NULL).
  - Return a pointer to your newly created node.

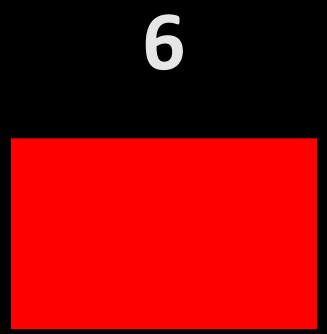
```
node *n = malloc(sizeof(node));
if (n == NULL) //not successful
{
    return false or 1 or any exit!
}

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

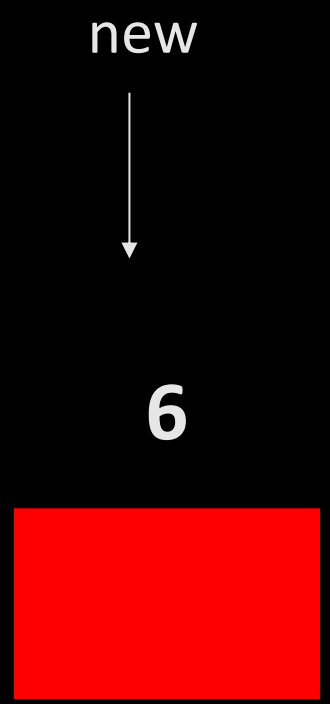
- Create a linked list:
  - Dynamically allocate space for a new (your first!) node.
  - Check to make sure you didn’t run out of memory.
  - Initialize the value field.
  - Initialize the next field (specifically, to NULL).
  - Return a pointer to your newly created node.

6

- Create a linked list:
  - Dynamically allocate space for a new (your first!) node.
  - Check to make sure you 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.

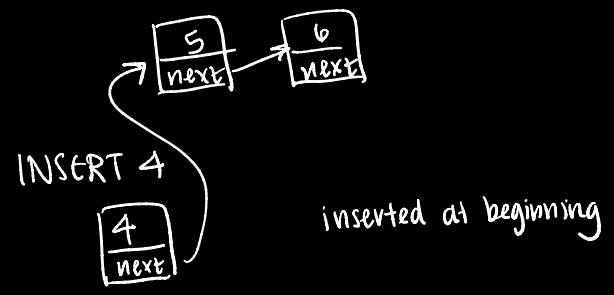


- Create a linked list:
  - Dynamically allocate space for a new (your first!) node.
  - Check to make sure you didn’t run out of memory.
  - Initialize the value field.
  - Initialize the next field (specifically, to NULL).
  - Return a pointer to your newly created node.



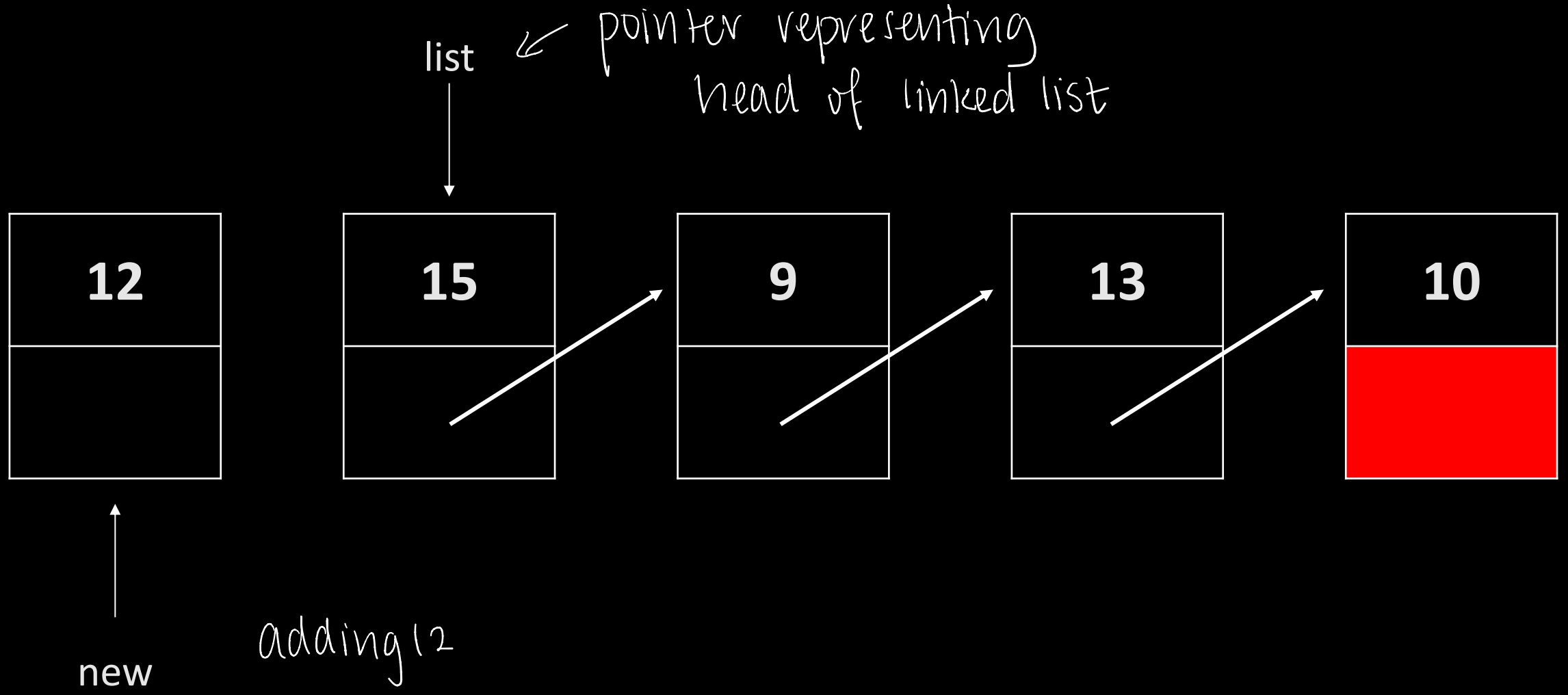
- Find an element:
  - Create a traversal pointer pointing to the list's head (first element).
  - If the current node's value field is what we're looking for, return true.
  - If not, set the traversal pointer to the next pointer in the list and go back to the previous step.
  - If you've reached the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

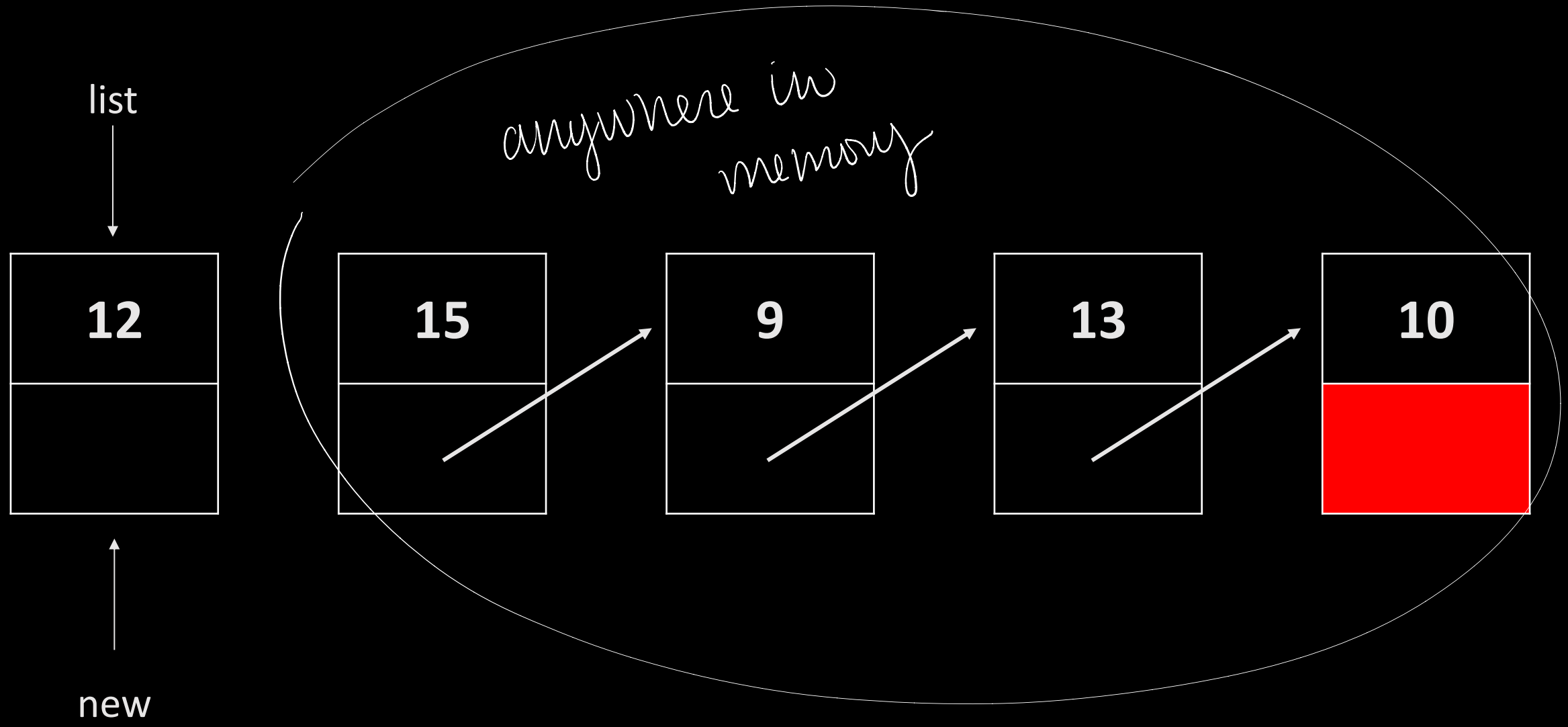
- Insert an element:
  - Dynamically allocate space for a new linked list node. //same thing as earlier
  - Populate and insert the node at the beginning of the linked list.
  - Return a pointer to the new head of the linked list.

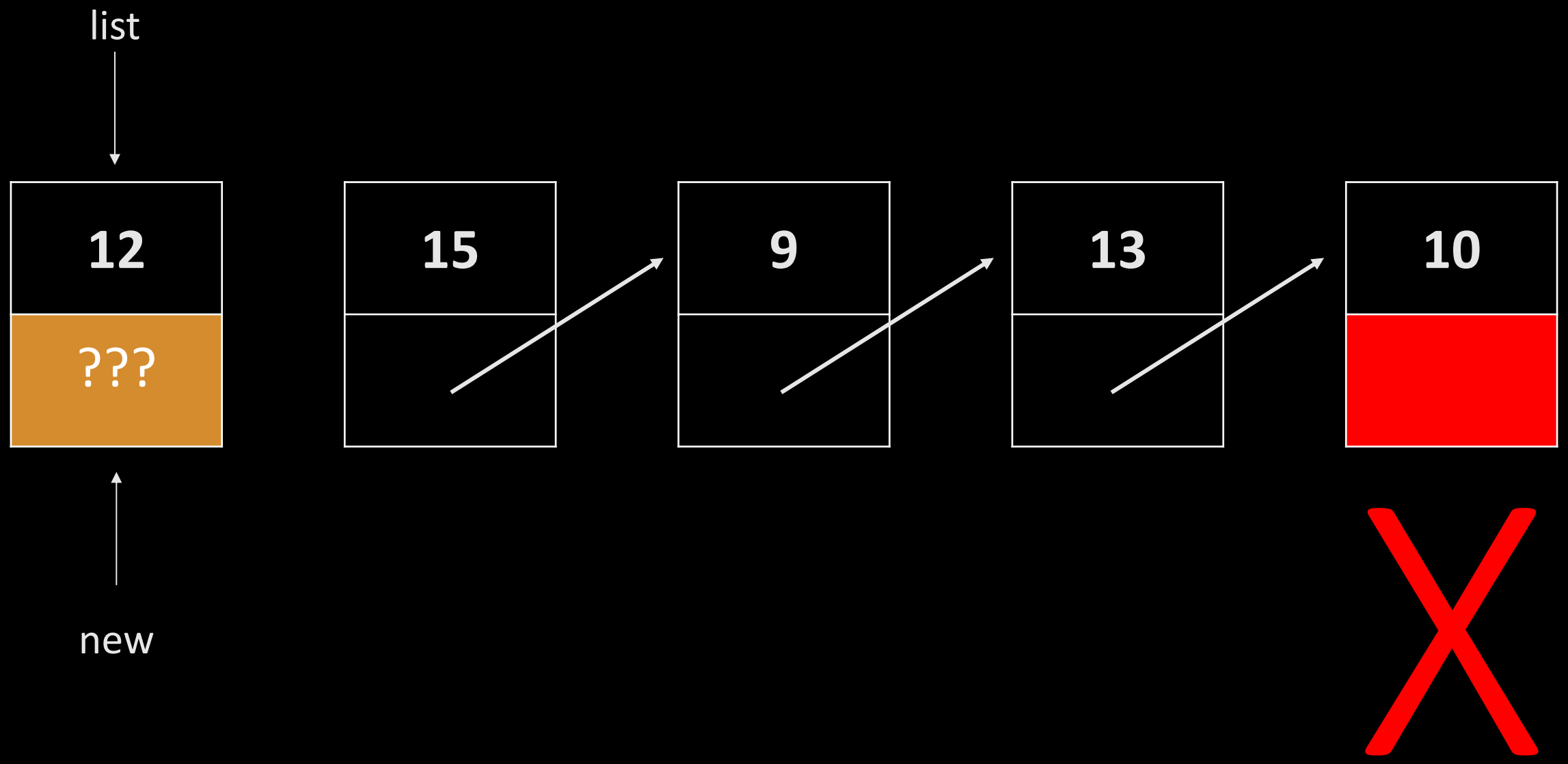


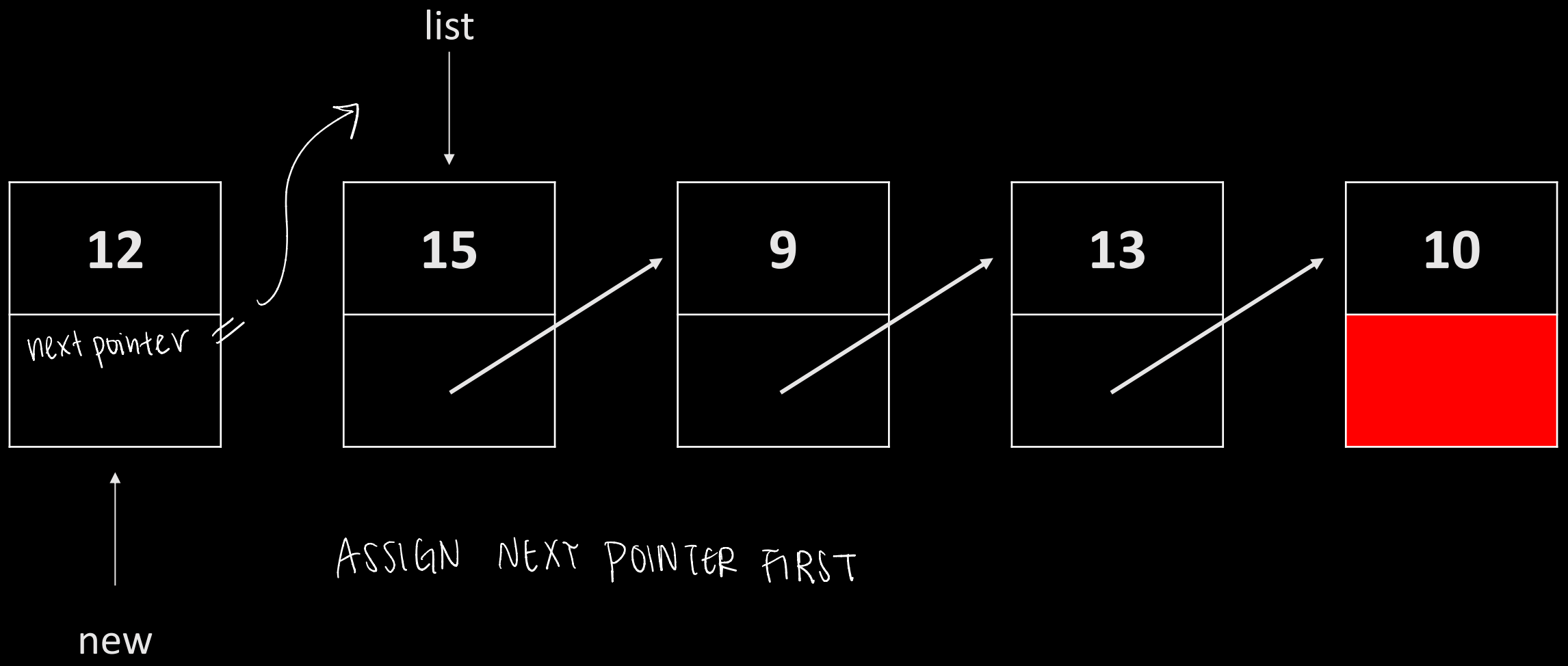
- Insert an element:
  - Dynamically allocate space for a new linked list node.
  - Check to make sure we didn't run out of memory.
  - Populate and insert the node at the beginning of the linked list.
    - So which pointer do we move first? The pointer in the newly created node, or the pointer pointing to the *original* head of the linked list?
    - This choice matters!
  - Return a pointer to the new head of the linked list.

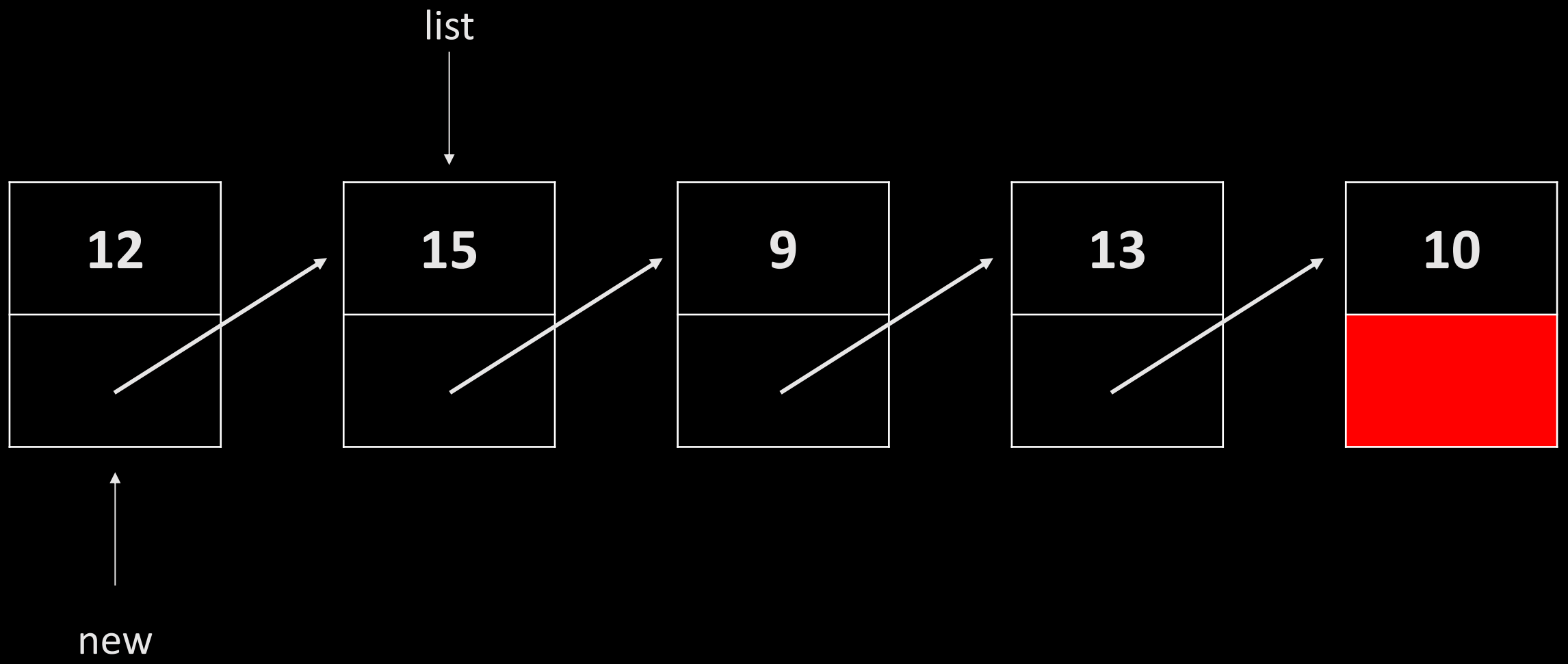
ORDER MATTERS!

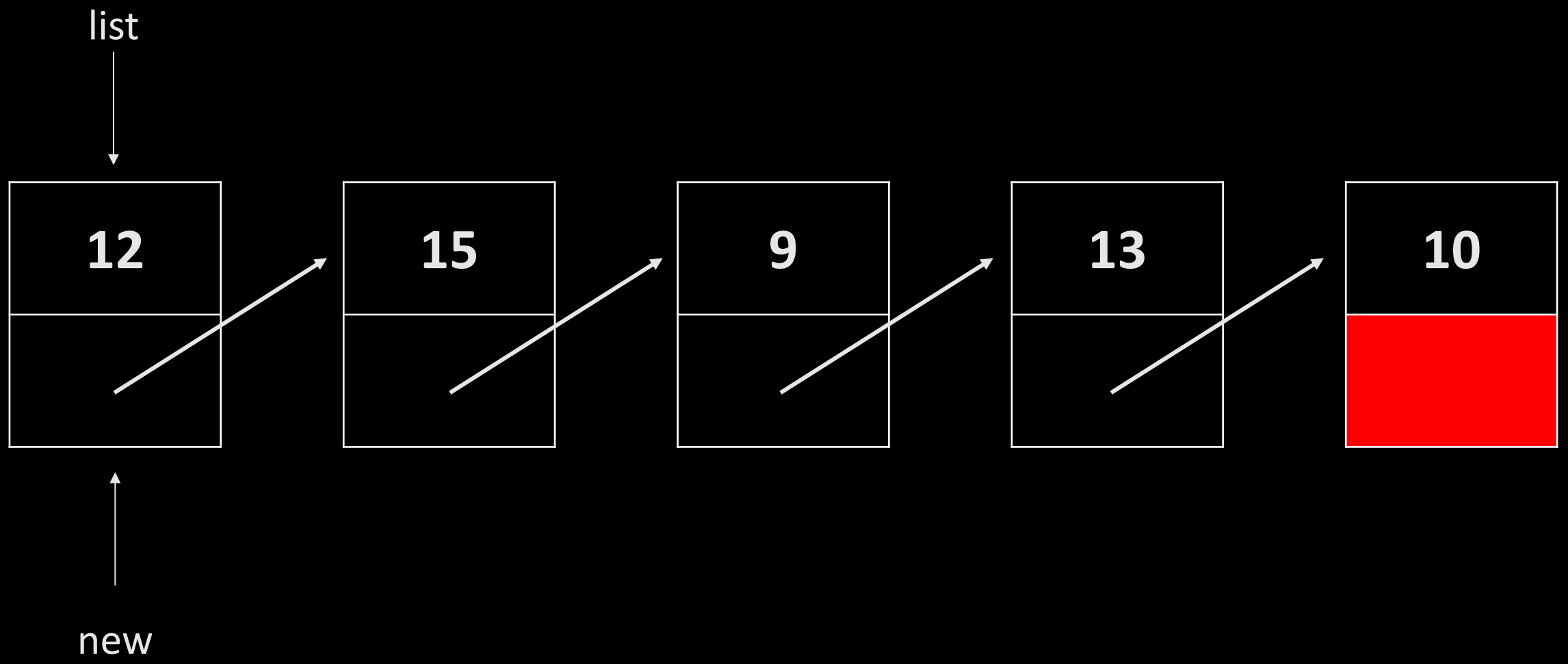

list
pointer representing
head of linked list
12
15
9
13
10
new
adding 12


list
anywhere in memory
12
15
9
13
10
new

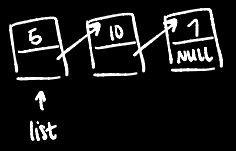

list
12
???
15
9
13
10
new


list
12
next pointer
15
9
13
10
new
ASSIGN NEXT POINTER FIRST


list
12
15
9
13
10
new


list
12
15
9
13
10
new

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



1. free (list) → only 5 is freed. 10 & 7 stuck somewhere

2. free (list→next) → 5 and 10 are freed
   free (list) → 7 is somewhere in memory

CORRECT WAY:

```
while (list != NULL)
{
    node *tmp = list;
    list = list→next;
    free (tmp);
}
```

list = 5
tmp = 5
list = 10
free (5)

list = 10
tmp = 10
list = 7
free (10)

list = 7
tmp = 7
list = NULL
free (7)

- Sounds like recursion!

Exercise

Write a program that prompts the user to type in integers, adds each integer one at a time to the head of a linked list, and then prints out the integers in the linked list (they'll be in reverse order from the input).

User input : 5, 4, 7

print : 7, 4, 5

User should stop typing in integers when 5 is inputted.

If you finish early, change the implementation to add a new node to the end of the list!

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

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

Struct for node!

```
int main(void)
{
    node *list = NULL;
    while (true)
    {
        int x = get_int("Number: ");
        if (x == 5)
        {
            printf("\n");
            break;
        }
        // TODO: Allocate a new node.
        // TODO: Add new node to head of linked list.
    }
// TODO: Print all nodes.   // 1 for loop, 1 line in for loop
// TODO: Free all nodes.
}
```

Exercise

```
*n. __ =
 n -> __ =
```

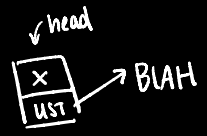
Allocate new node:

```
node *n = malloc(sizeof(node));
// check if we're out of memory!
n -> number = x;
n -> next = NULL;
```

list

Add this node!

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



Print

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

Free

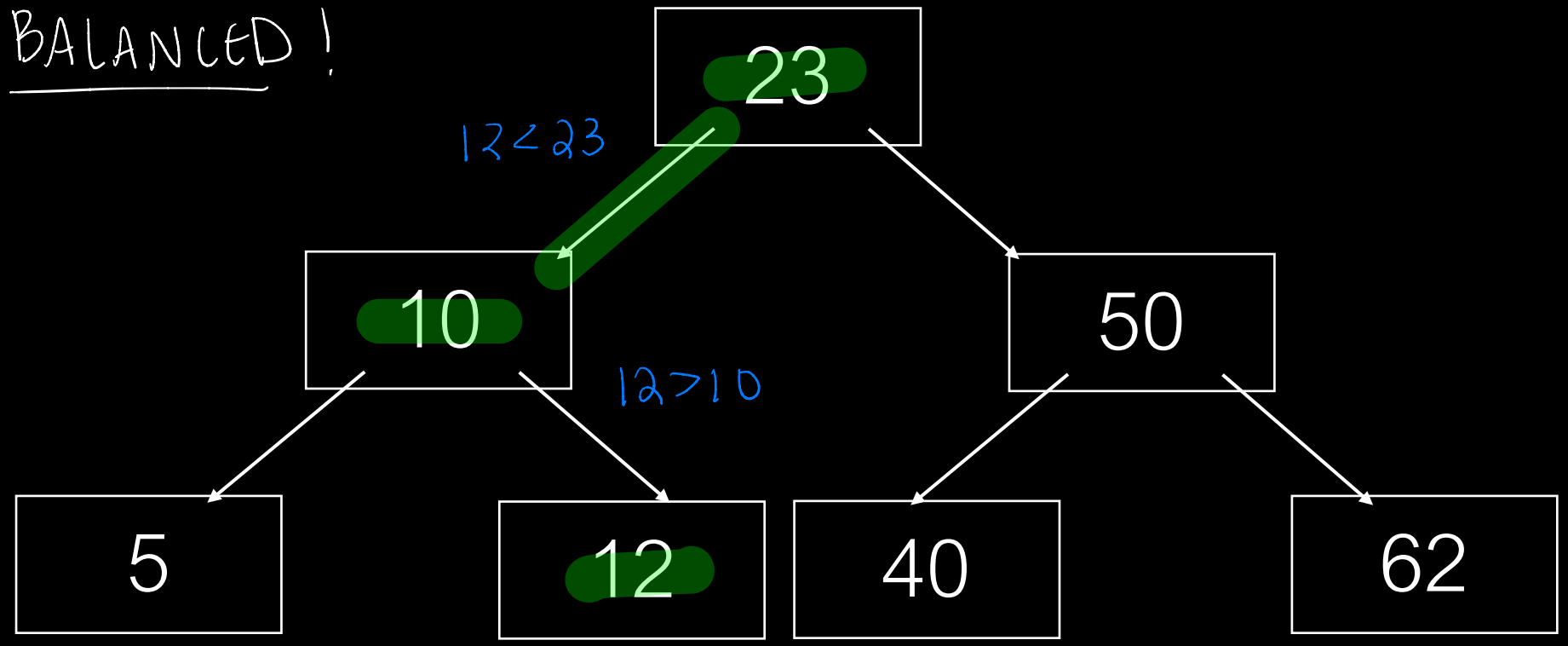
```
node *ptr = list;
while (ptr != NULL)
{
    node *tmp = ptr;
    ptr = ptr -> next;
    free(tmp);
}
```

Binary Search Trees

Binary Search Tree

- Let's turn this list into a data structure!

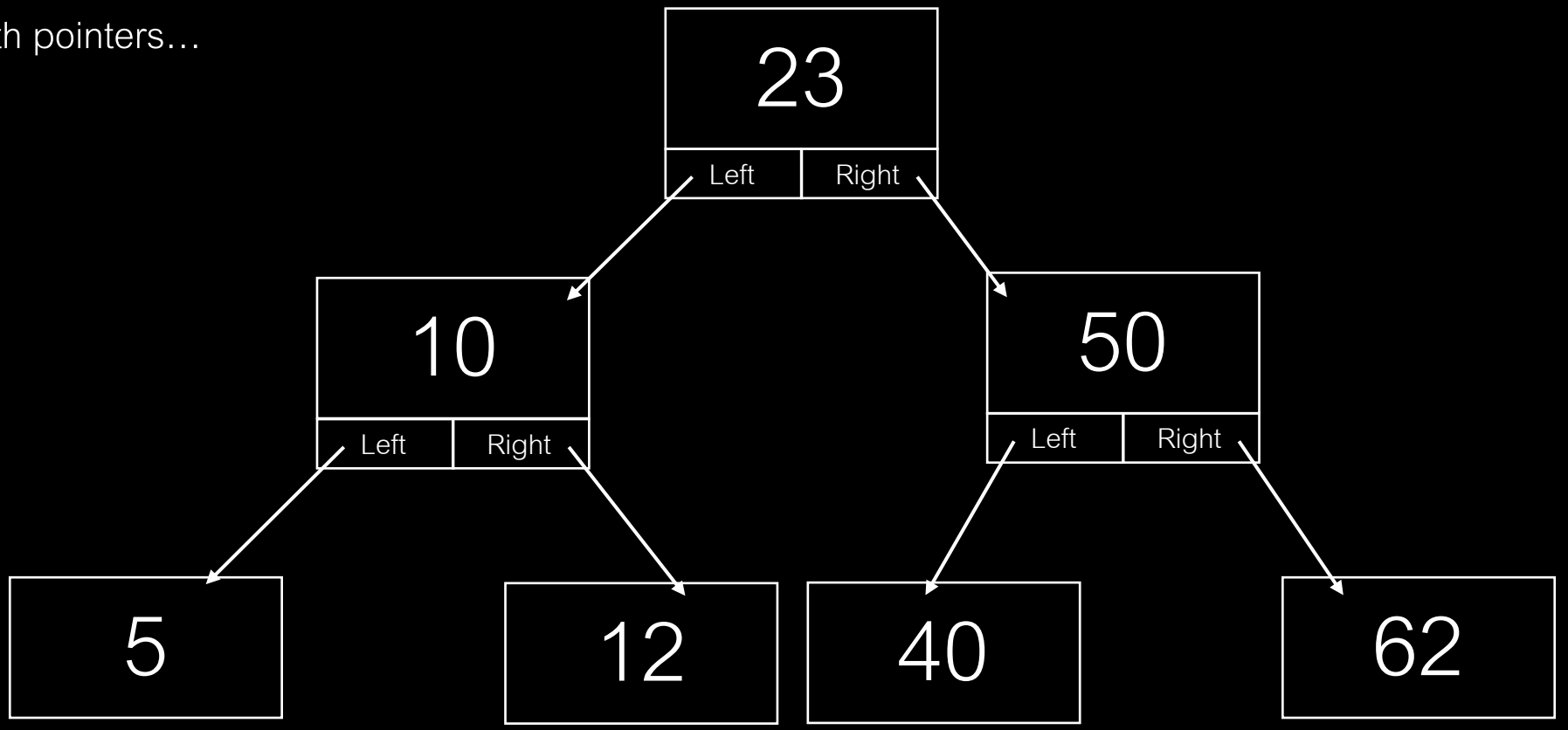
5	10	12	23	40	50	62



Binary Search Tree

nodes have
2 pointers:
left right

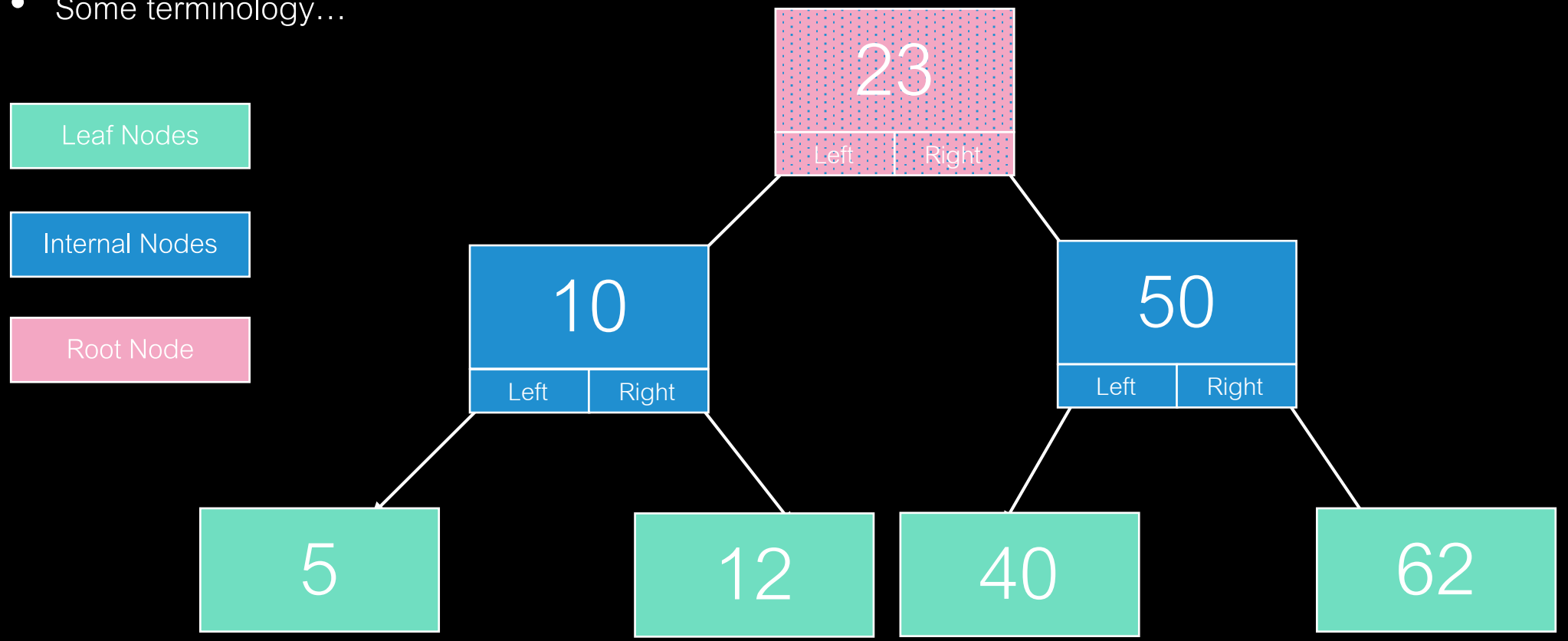
- With pointers…



- At each memory address, there is a node containing a value, an address to the left child, and an address to the right child.

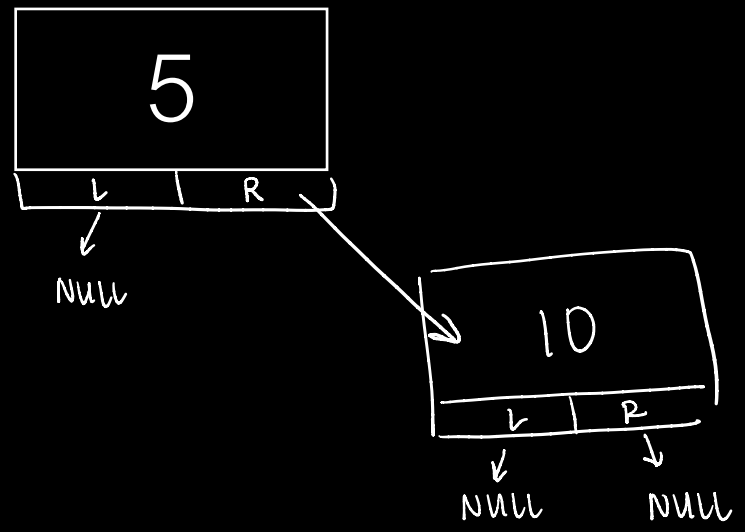
Binary Search Tree

- Some terminology…



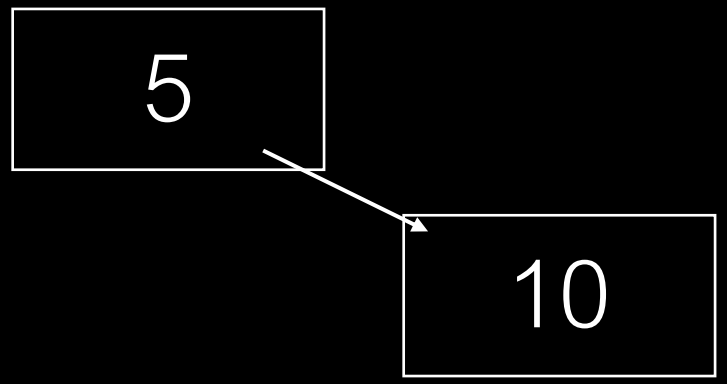
Binary Search Tree – Insertion

- Now to create a binary search tree one node at a time from these numbers… 5, 10, 30, 50



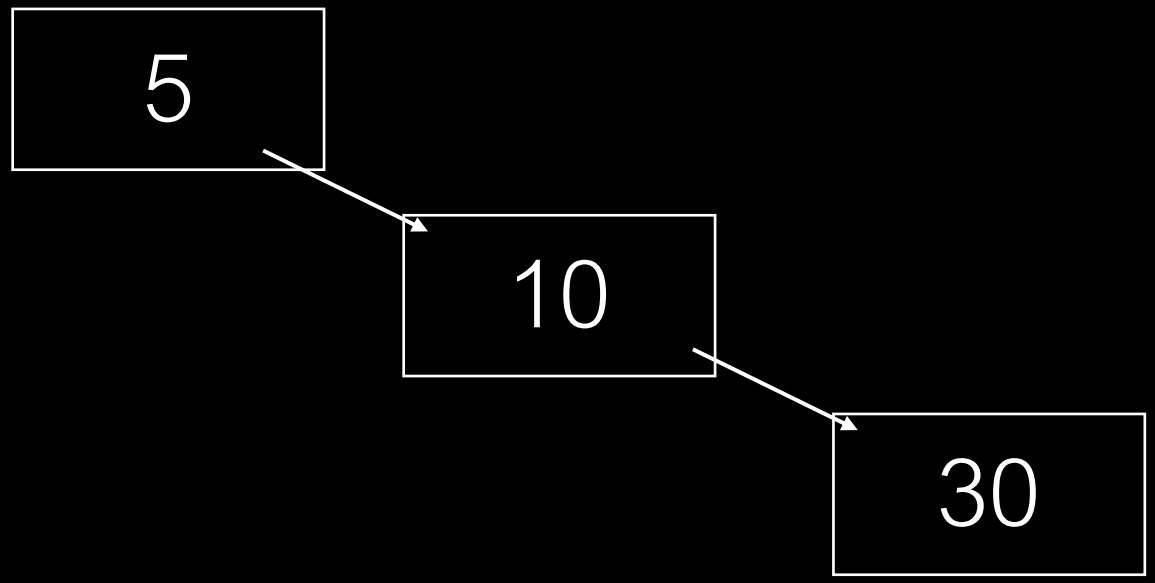
Binary Search Tree – Insertion

- Now to create a binary search tree one node at a time from these numbers… 5, 10, 30, 50



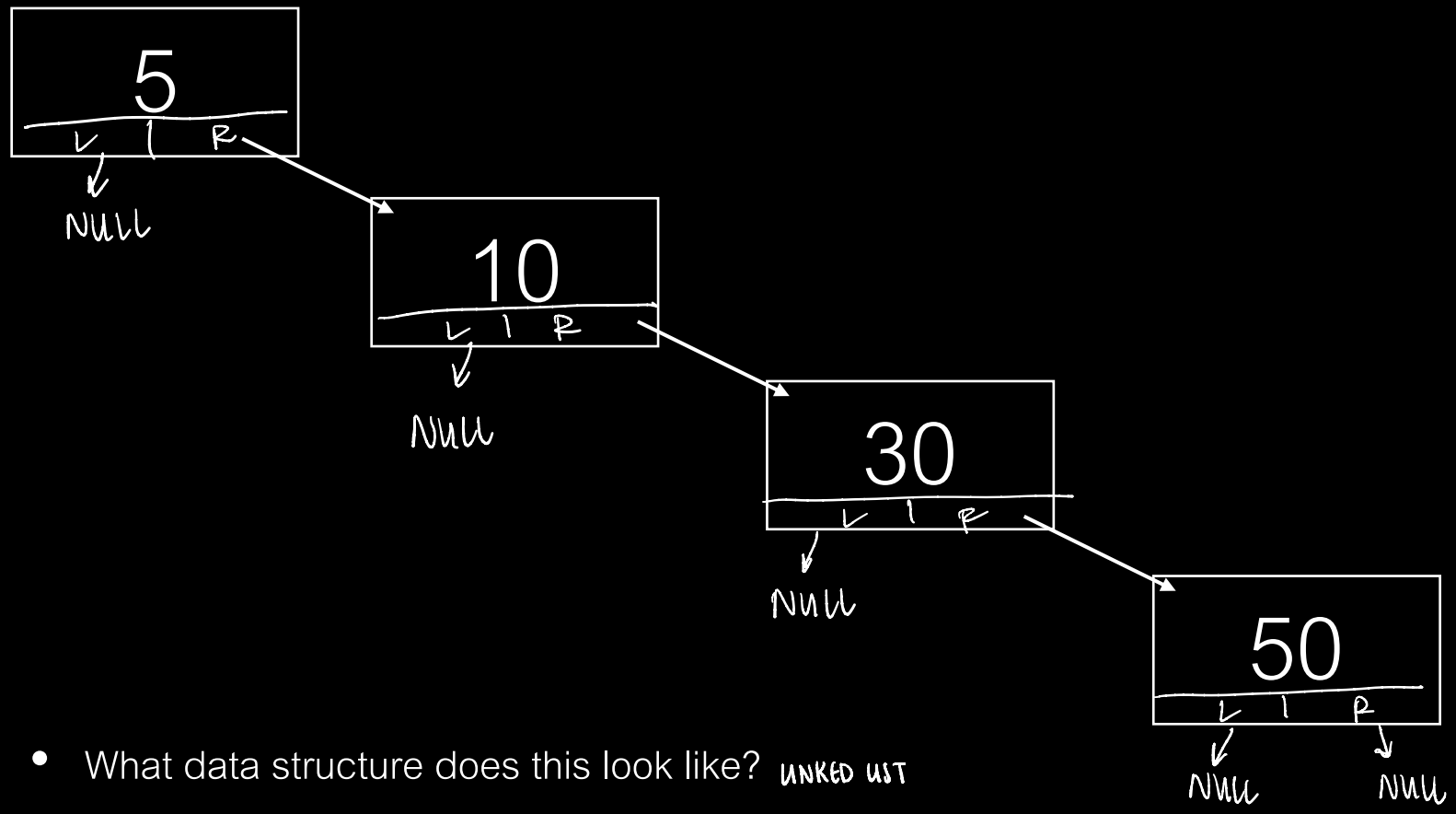
Binary Search Tree – Insertion

- Now to create a binary search tree one node at a time from these numbers… 5, 10, 30, 50



Binary Search Tree – Insertion

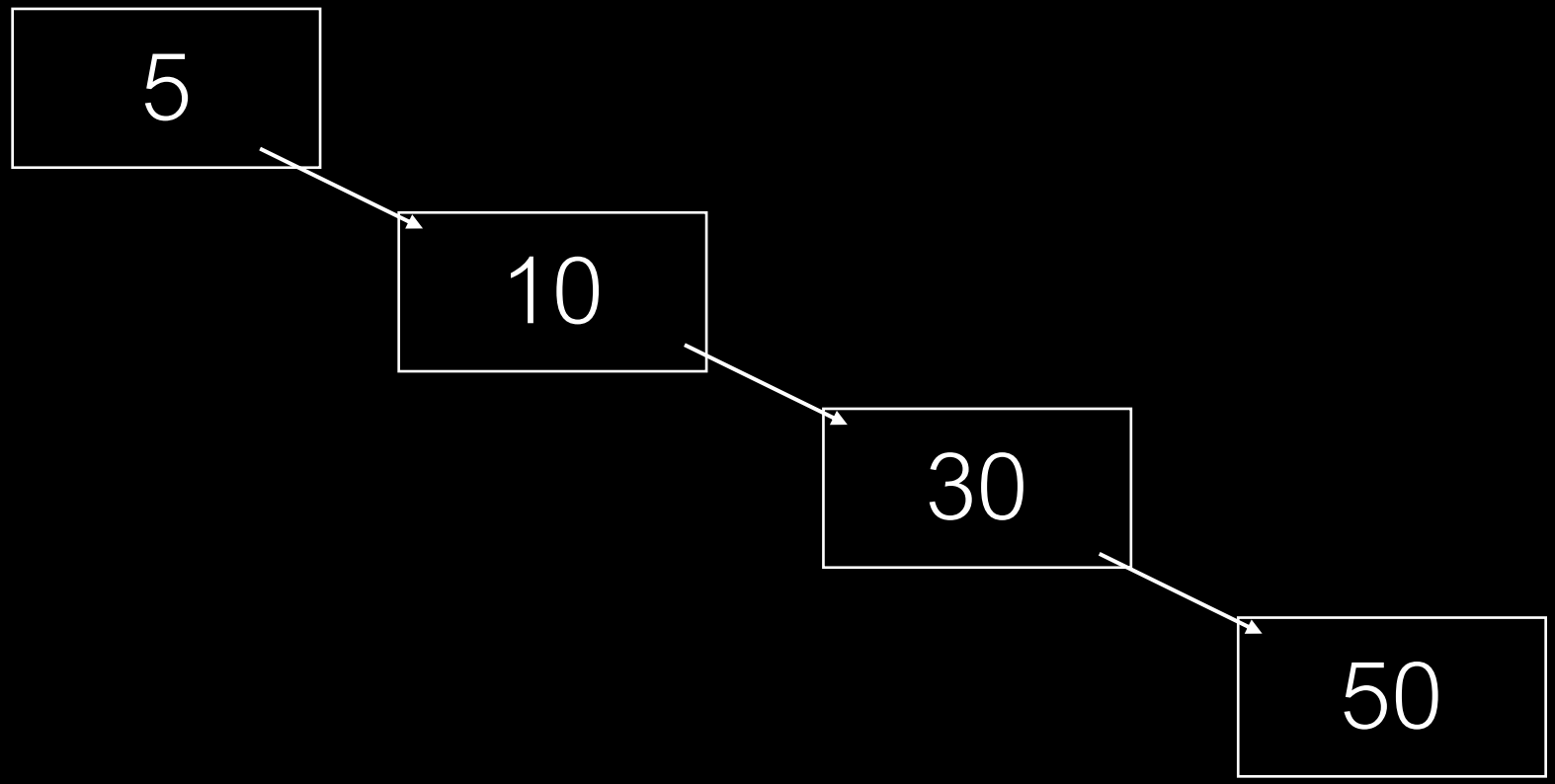
- Now to create a binary search tree one node at a time from these numbers… 5, 10, 30, 50



- What data structure does this look like? LINKED LIST

Binary Search Tree

- Let's search for 50… traverse through all nodes



Efficiency

Binary Search Tree

- Searching:

Efficiency

Binary Search Tree

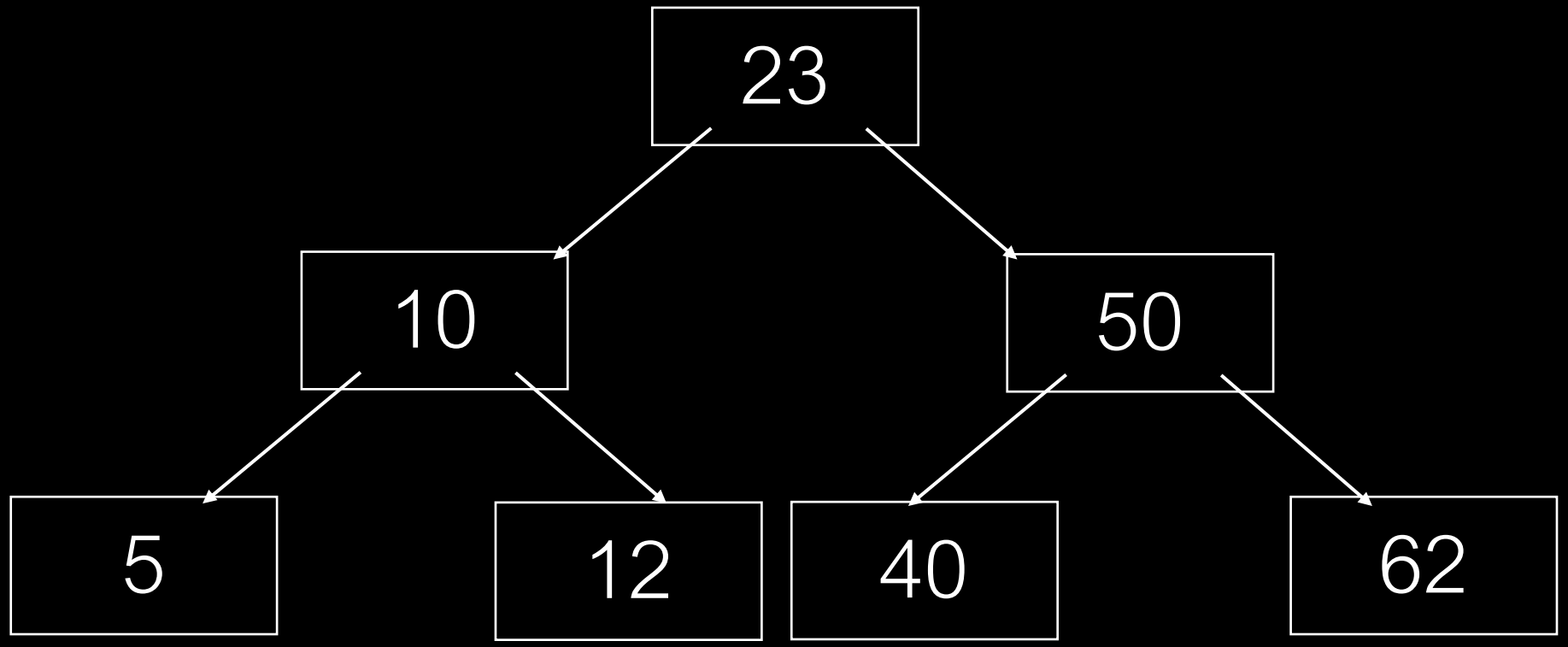
- Searching: O(n)
- Insertion/Deletion:

Efficiency

Binary Search Tree

- Searching: O(n)
- Insertion/Deletion: O(n)

Efficiency



Efficiency

Binary Search Tree

- Searching: O(n)
- Insertion/Deletion: O(n)

Balanced Binary Search Tree

- Searching:
- Insertion/Deletion:

Efficiency

Binary Search Tree

- Searching: O(n)
- Insertion/Deletion: O(n)

Balanced Binary Search Tree (AVL, Red Black)

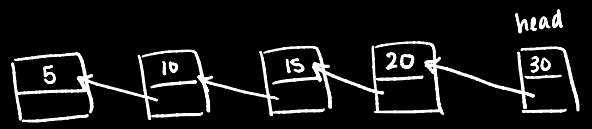
- Searching: O(log n)
- Insertion/Deletion: O(log n)

Stacks : abstract data structures

↳ arrays, linked lists

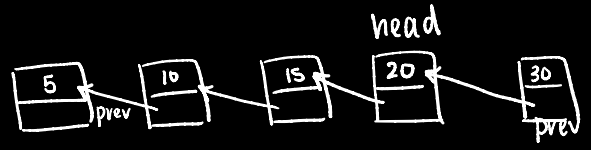
- last In, First Out

5, 10, 15, 20



instead of next, prev

remove 30

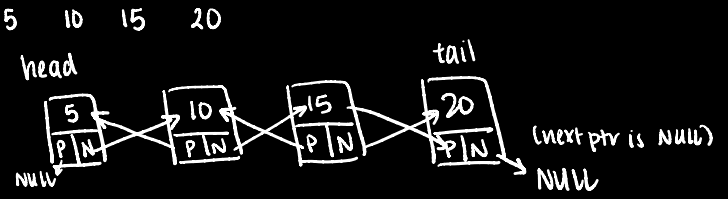



```
node *n = head
printf("%i\n", head->value);    30
head = head -> prev
free(n);
```

Queue : abstract data type

↳ LINKEDLIST

- line : First In, First Out



- doubly linked ⇒ linked on both sides (2 pointers)
- head & tail

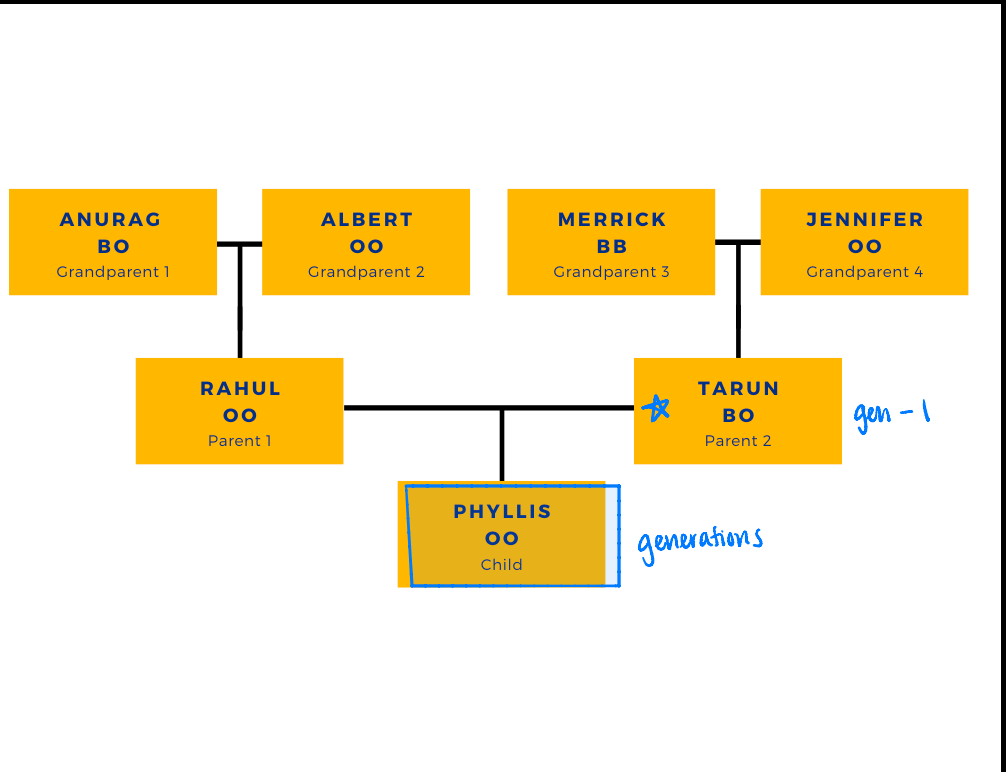
☆ node *n = head
printf("%i\n", head → value)
head = head → N
head → P = NULL
free(n)

QUEUE

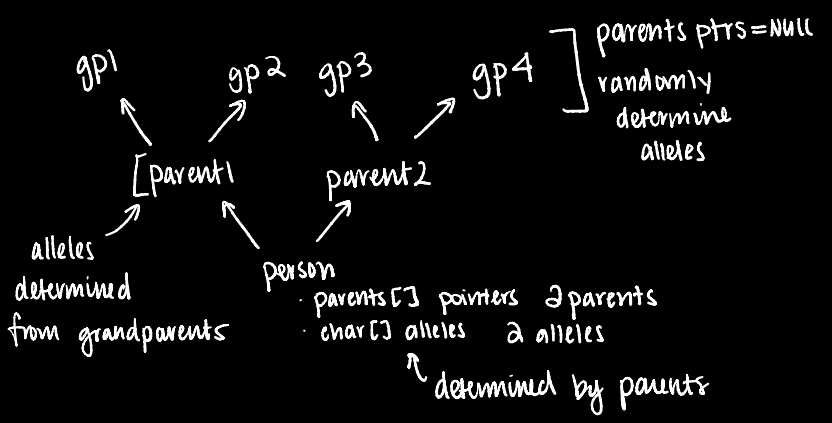
☆ node *n = tail
printf("%i\n", tail → value)
tail = tail → P
tail → N = NULL
free(n)

STACK

Lab



3 generation family



Hashtables

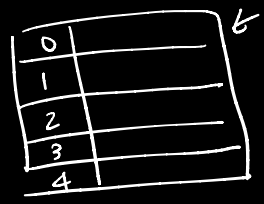
fscanf(____ , "%s", ____)
file dest

read the next string from file into your destination

table is an array!

table[index]

↑ hashcode



← linkedlist

malloc(sizeof(____))

check if it was successful

put new node into table

if at table[0] : linkedlist

node → next = table[0]

table[0] = node

~~Hashtables~~ LAB :)

person
*parents[2]
alleles[2]

create_family(int generations)

person *child = allocate memory for a type person REST: create child's parents & alleles

if (generations > 1) // gen. parents! until correct # gens

child->parent[0] = create_family(_____) //create a child's parent's family

//same thing for parent[1]

child -> alleles[0] = // random allele from parent rand()%2 → random int 0 or 1

//same thing for alleles[1]

else

//set child's parents to NULL!

//same for parent[1]

//set child's allele[0] to a random allele using random_allele() - gives random allele

//same for alleles[1]

free_family(person *p)

if(___) //base case

{ return; }

//free the families of the parents

free(p)

}