C++ MEMORY MODEL
DYNAMIC MEMORY
HEAP VS STACK

Problem Solving with Computers-I

Announcements

Please fill out the mid-quarter evaluations for:

(1) TAs: http://bit.ly/CS16-Midquarter-TA-Evaluation

(2) Course and instructor:
http://bit.ly/CS16-Midquarter-Instructor-and-Course-Evaluation

Link to both is posted on Piazza!

The case of the disappearing data!

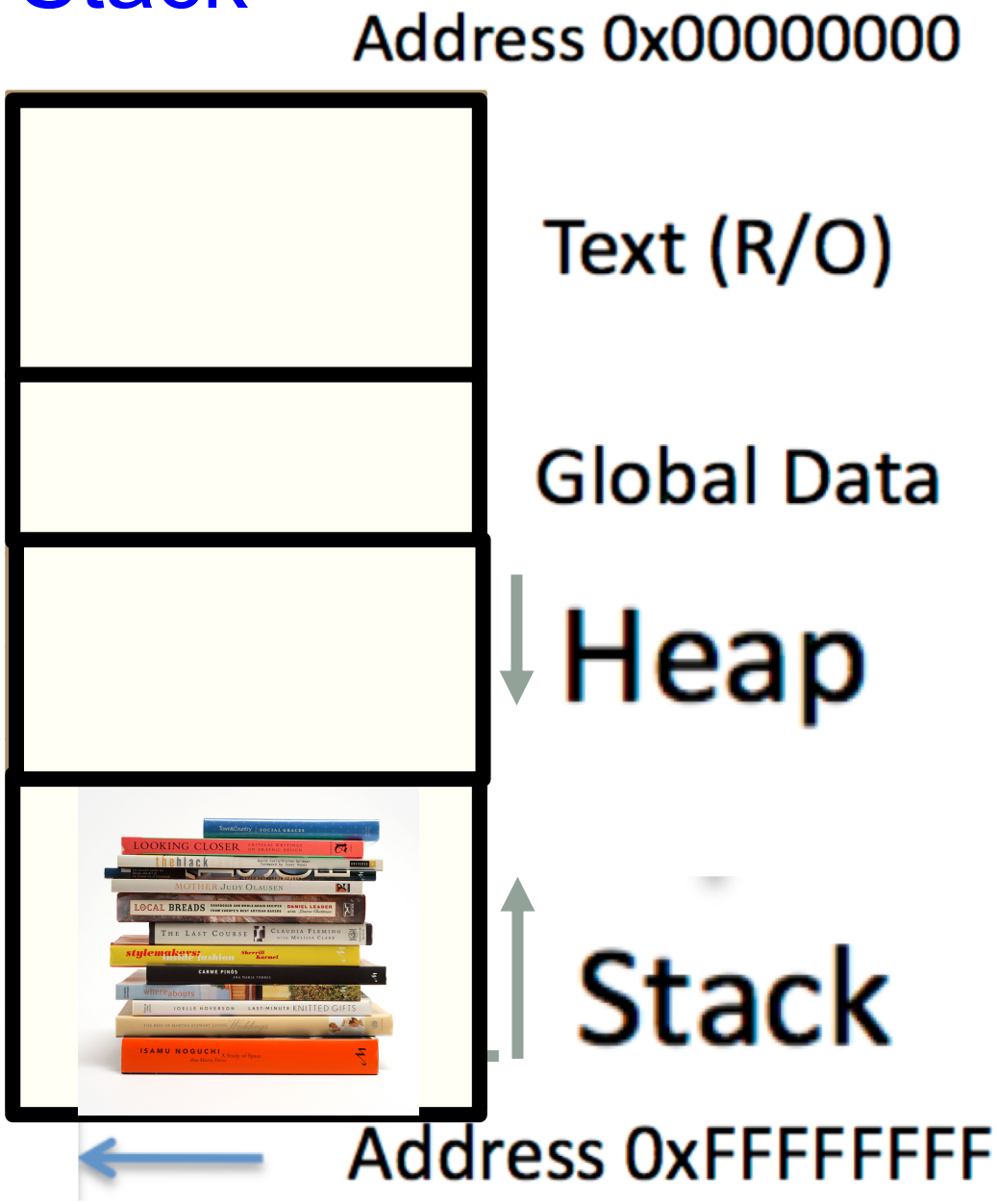
```
int getInt(){
      int x=5;
      return x;
}
int* getAddressOfInt(){
      int x=10;
      return &x;
}
int main(){
      int y=0, *p=nullptr, z=0;
      y = getInt();
      p = getAddressOfInt();
      z = *p;
     cout<<y<<", "<<z<<", "<<*p<<endl;
}
```

What is the output?

A. 5, 0, 10
B. 5, 10, 10
C. Something else

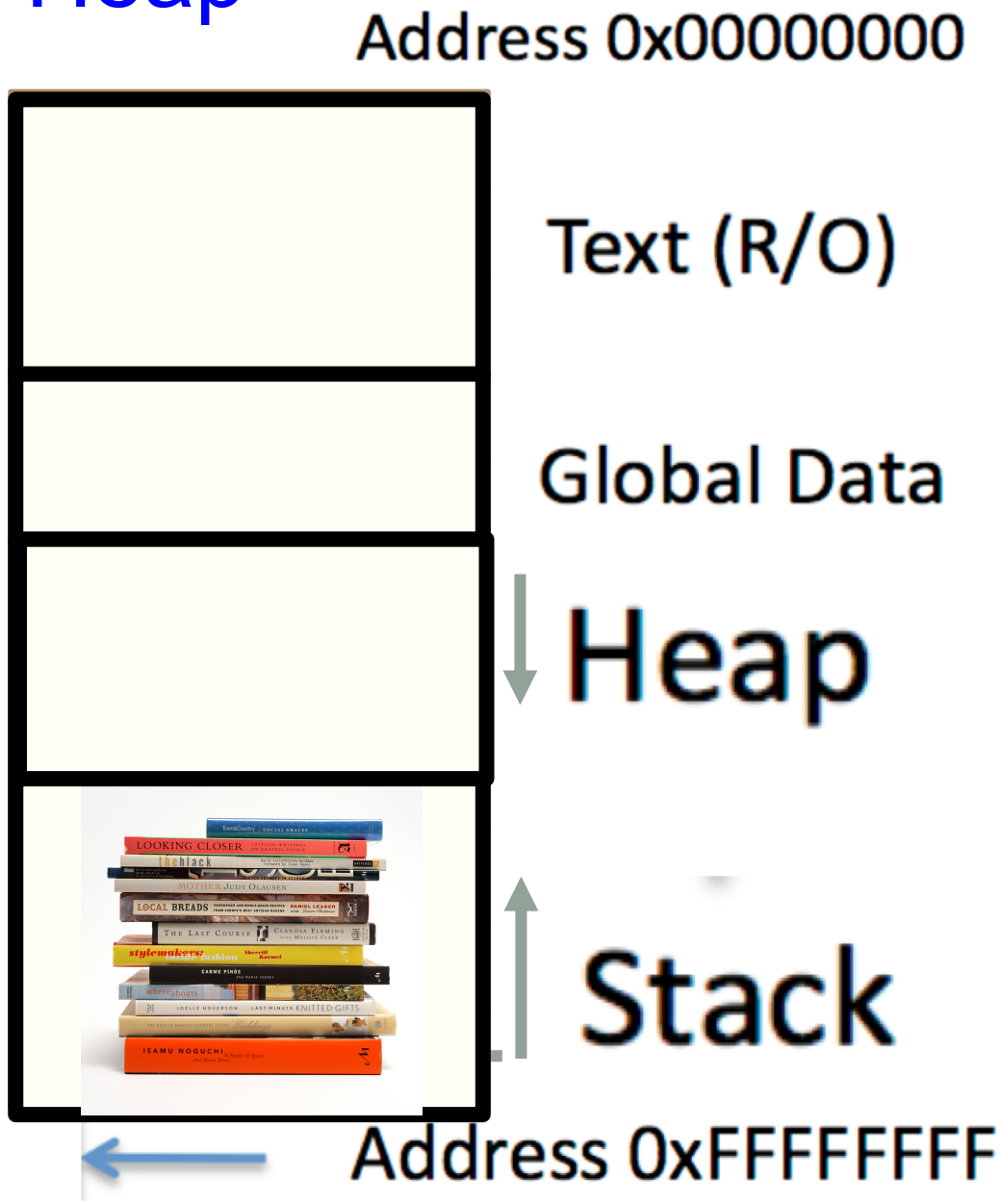
C++ Memory Model: Stack

- Stack: Segment of memory managed automatically using a Last in First Out (LIFO) principle
- Think of it like a stack of books!



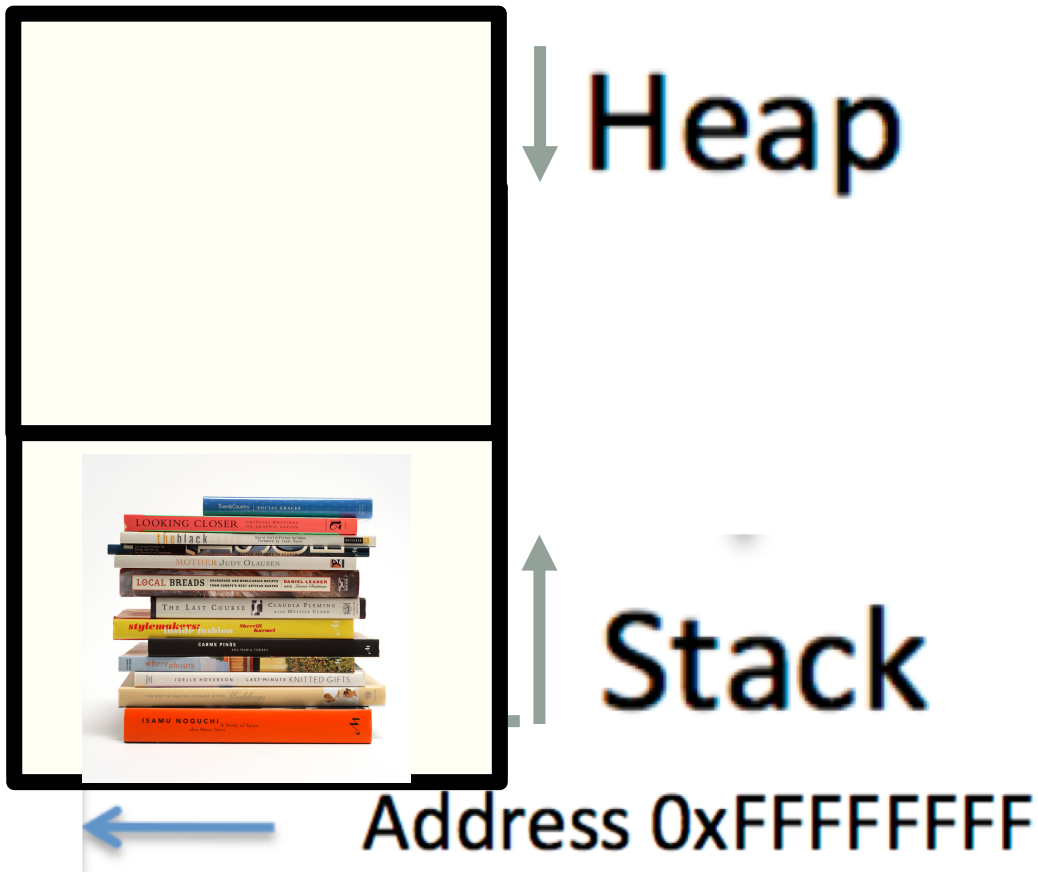
C++ Memory Model: Heap

- Heap: Segment of memory managed by the programmer
- Data created on the heap stays there
  - FOREVER or
  - until the programmer explicitly deletes it



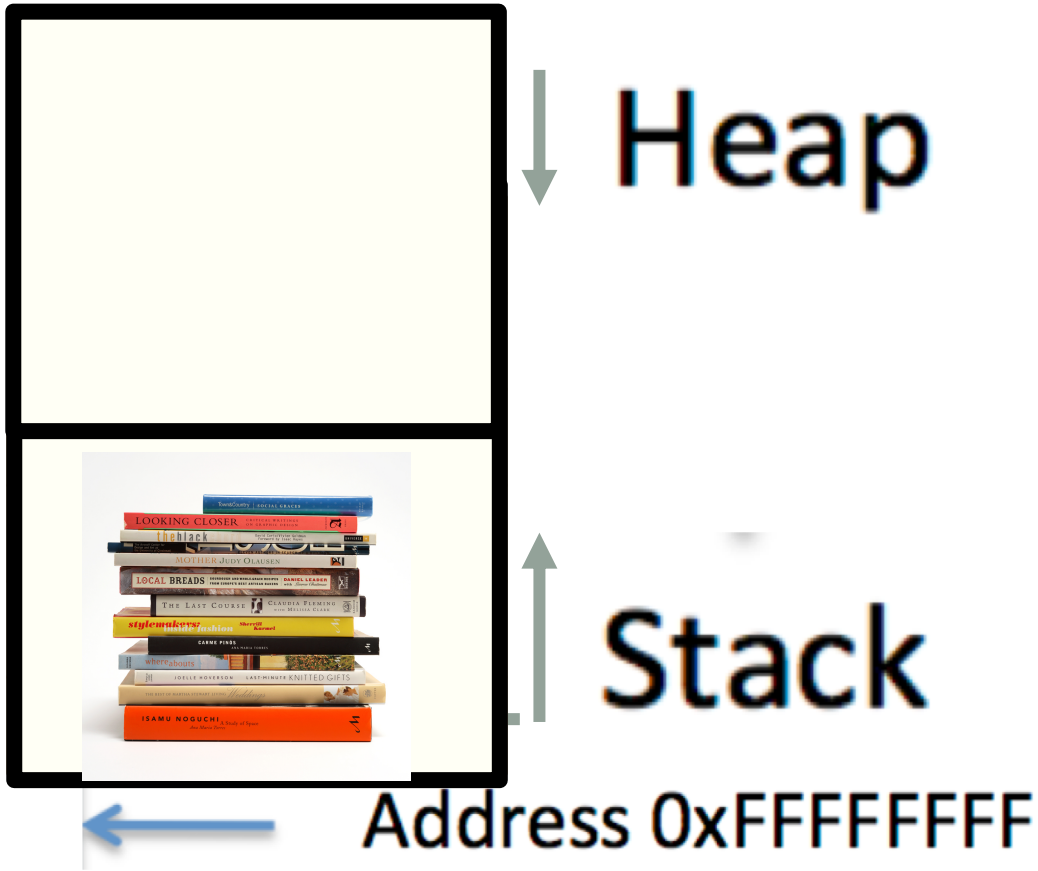
Creating data on the Heap: new

To **allocate** memory on the heap use the **new** operator



Deleting data on the Heap: delete

To **free** memory on the heap use the **delete** operator



Dynamic memory management = Managing data on the heap

```
int* p= new int; //creates a new integer on the heap

Student* n = new Student;

                    //creates a new Student on the heap

delete p; //Frees the integer

delete n; //Frees the Student
```

Solve the case of the disappearing data!

```
int getInt(){
      int x=5;
      return x;
}
int* getAddressOfInt(){
      int x=10;
      return &x;
}
int main(){
    int y=0, *p=nullptr, z=0;
    y = getInt();
    p = getAddressOfInt();
    z = *p;
    cout<<y<<", "<<z<<", "<<*p<<endl;
}
```

Change the code so that *p does not disappear

Desired output:
5, 10, 10

Heap vs. stack

```
1 #include <iostream>
2 using namespace std;
3
4 int* createAnIntArray(int len){
5
6     int arr[len];
7     return arr;
8
9 }
```

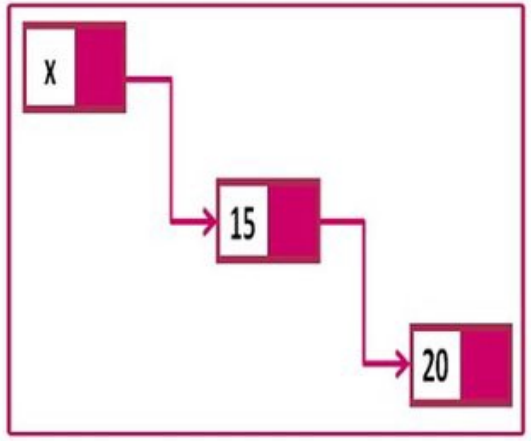
Does the above function correctly return an array of integers?

A. Yes

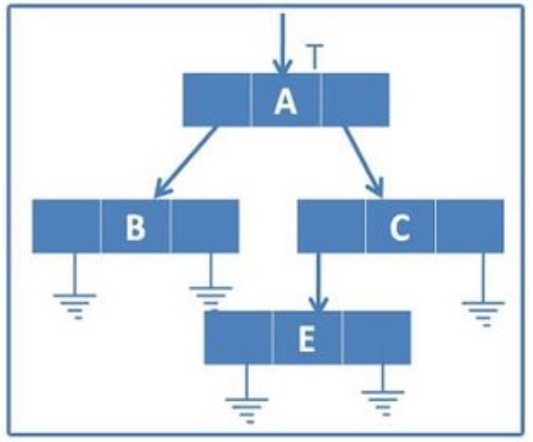
B. No

Where are we going? Data Structures!

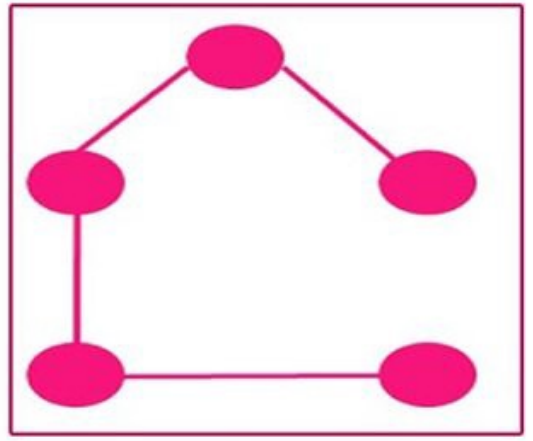
Arrays



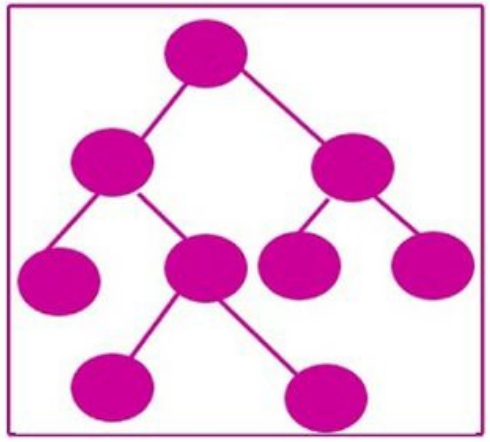
Link list



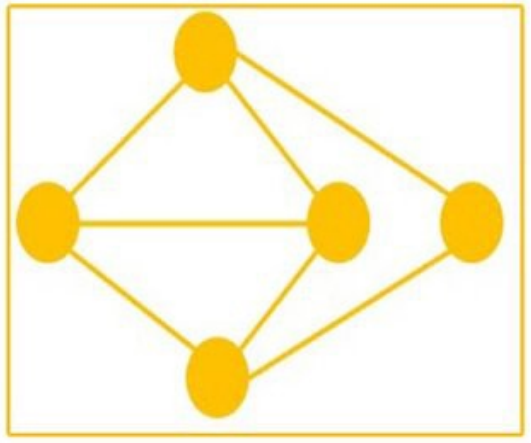
list



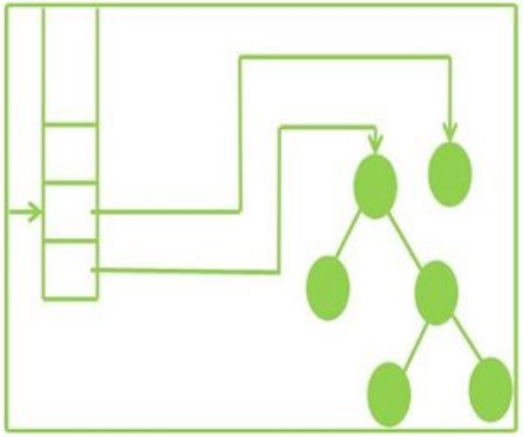
spanning tree



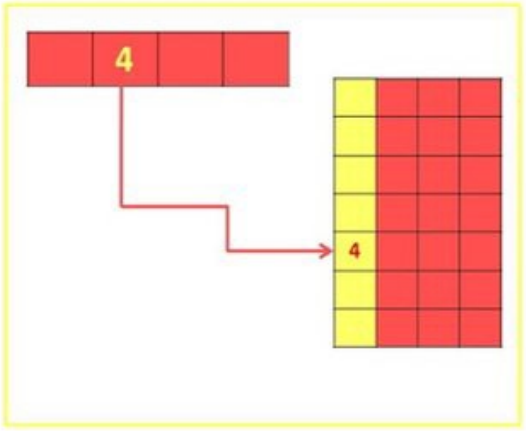
Tree



Graph



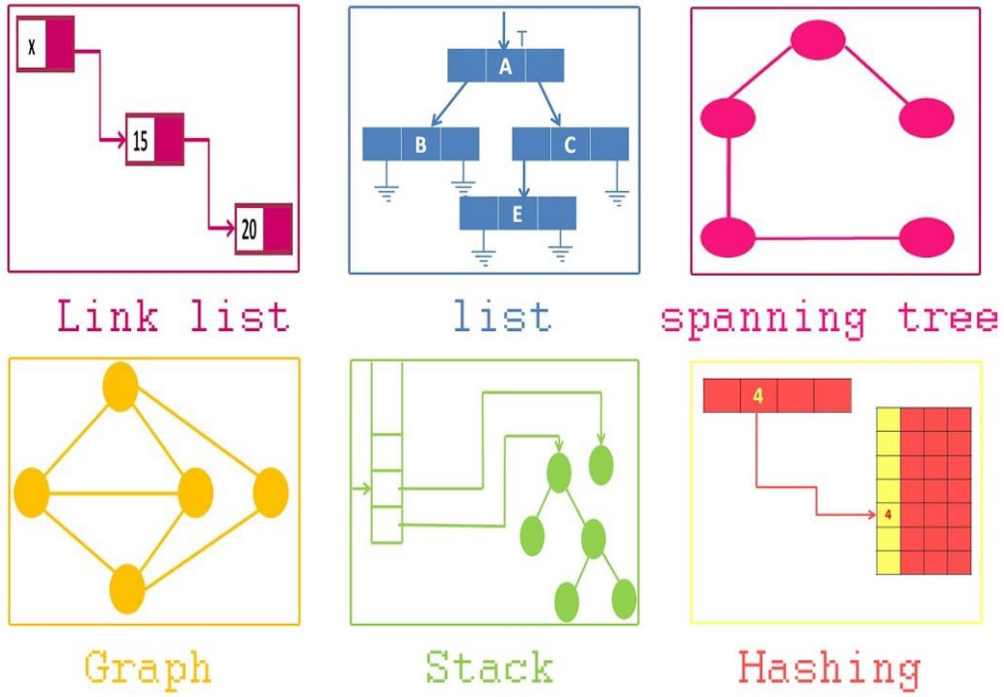
Stack



Hashing

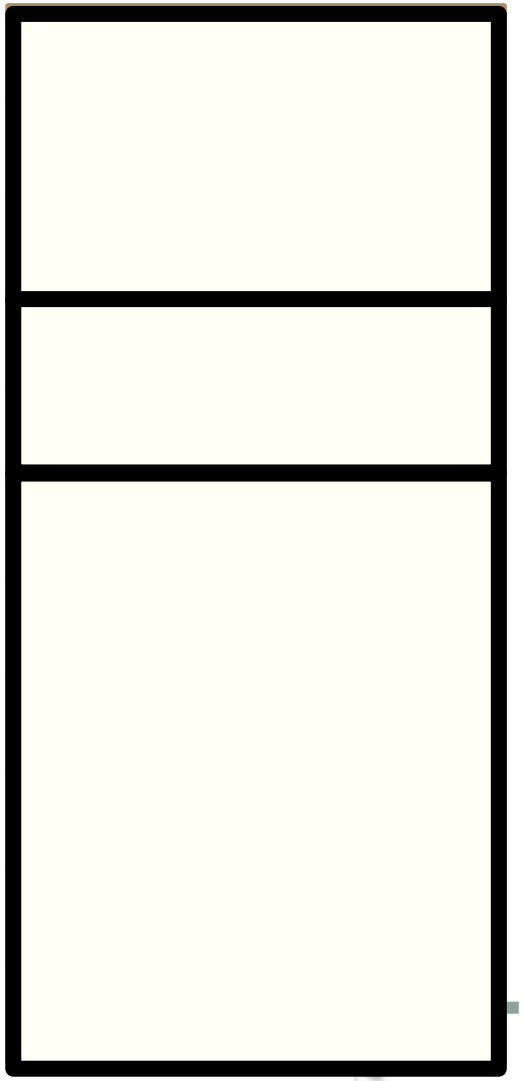
Where are we going? Data structures!!

Arrays



(CS24/32)

It all boils down to 1's and 0's



Address 0x00000000

Text (R/O)

Global Data

Heap

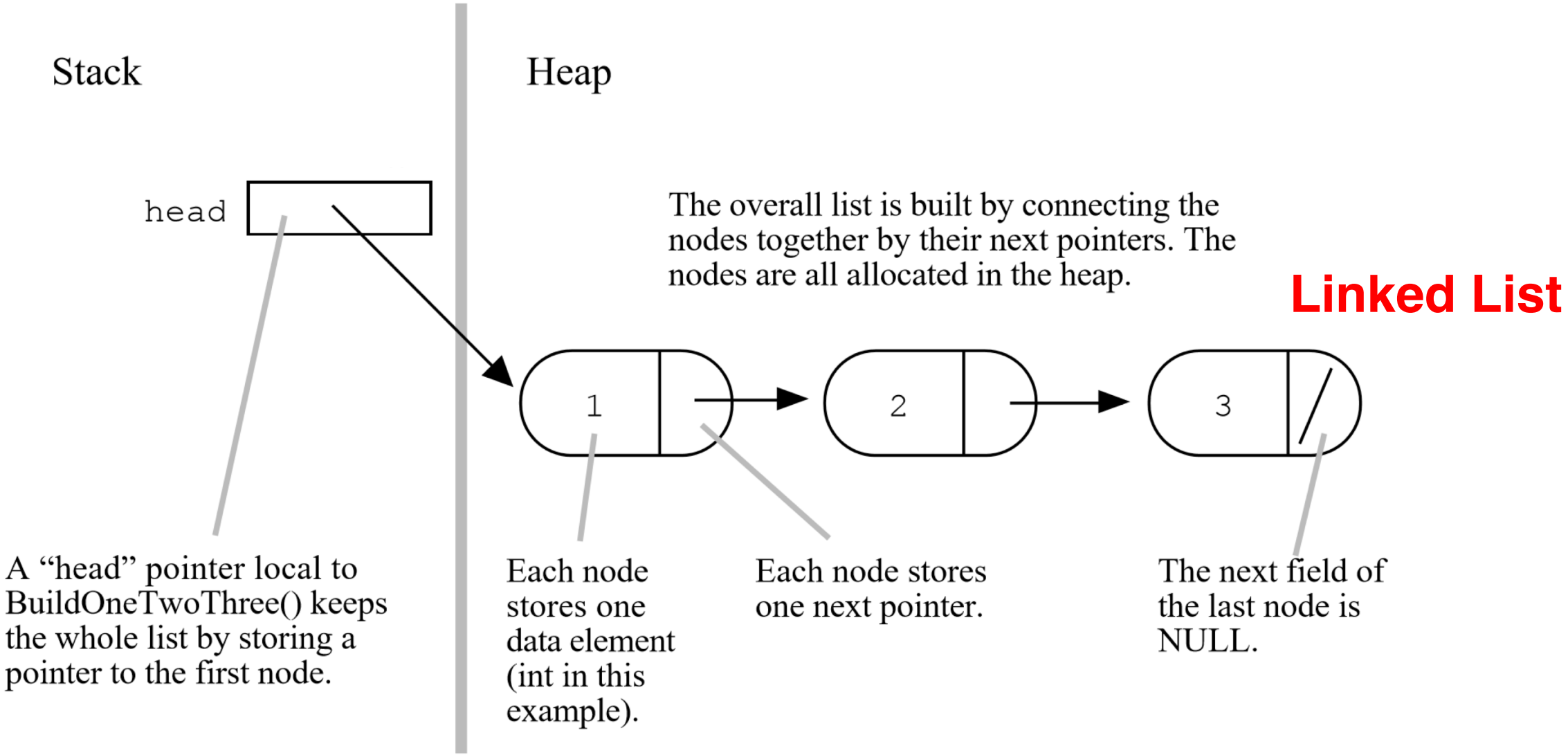
Stack

Address 0xFFFFFFFF

Linked Lists

Array List

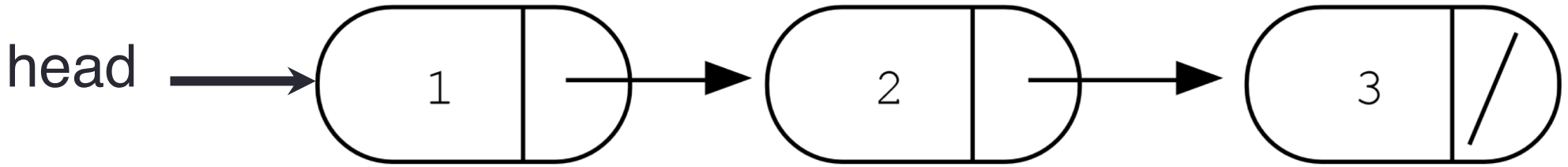
The Drawing Of List {1, 2, 3}



Linked List

Accessing elements of a linked list

```
struct Node {
      int data;
      Node *next;
};
```



Assume the linked list has already been created, what do the following expressions evaluate to?

1. head->data
2. head->next->data
3. head->next->next->data
4. head->next->next->next->data

A. 1
B. 2
C. 3
D. NULL
E. Run time error

Create a small list – use only the stack

- Define an empty list
- Add a node to the list with data = 10

```
struct Node {
        int data;
        Node *next;
};
```

Heap vs. stack

```
Node* createSmallLinkedList(int x, int y){

    Node* head = NULL;
    Node n1 ={x, NULL};
    Node n2 ={y, NULL};
    head = &n1;
    n1->next = &n2;
    return head;
}
```

Does the above function correctly create a two-node linked list?

A. Yes
B. No

Pointer pitfalls and memory errors

- **Segmentation faults**: Program crashes because it attempted to access a memory location that either doesn't exist or doesn't have permission to access
- Examples of code that results in undefined behavior and potential segmentation fault

```
int arr[] = {50, 60, 70};

for(int i=0; i<=3; i++){
   cout<<arr[i]<<endl;
}
```

```
int x = 10;
int* p;
cout<<*p<<endl;
```

Dynamic memory pitfalls

Dangling pointer: Pointer points to a memory location that no longer exists

Which of the following functions returns a dangling pointer?

```
int* f1(int num){
    int* mem1 =new int[num];
    return(mem1);
}
```

```
int* f2(int num){
     int mem2[num];
     return(mem2);
}
```

A. f1
B. f2
C. Both
D. Neither

Dynamic memory pitfalls

Memory leaks (tardy free):

Heap memory not deallocated before the end of program

Heap memory that can no longer be accessed

Example

```
void foo(){
     int* p = new int;

}
```

Next time

- More Linked Lists