CS106B Final Review

Ashley Taylor, Zoe Robert

Today’s Session Overview

● Logistics	Anton
● Classes and Structs	Ashley
● Sorting	
● Pointers	
● Memoization	
● Linked Lists	
● Hashing	
● Trees	Zoe
● Binary Heap	
● Binary Search Trees	
● Graphs	
● BFS and DFS	
● Dijkstra and A*	
● Minimum Spanning Tree	

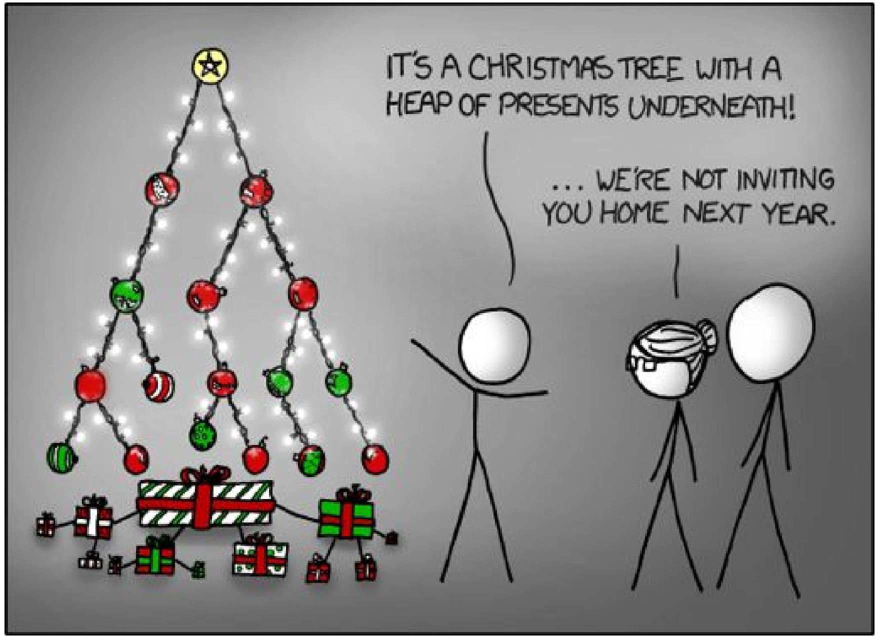
Logistics

Final Logistics

- Monday December 12, 8:30-11:30am
- Split into two locations based on last name:
    - A-G: Cubberley Auditorium
    - H-Z: Dinklespiel Auditorium
- Open books, open notes, open mind (but closed computer)
- Pencils and pens accepted

What’s on the final

- Everything is fair game!
- Will be weighted to second half of the course and on material done in assignments


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.

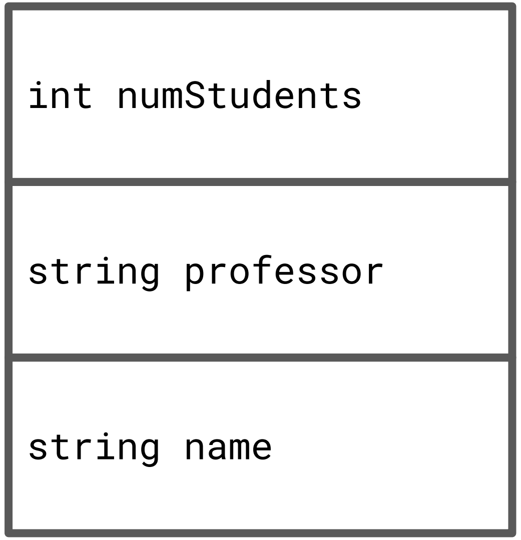
Classes/Structs!

Classes/Structs

Structs have fields, a good way to bundle information together

```
struct Course {
  int numStudents;
  string professor;
  string name;
};

Course cs106b;
cs106b.numStudents = 300;
cs106b.professor = “Chris”;
```



Classes/Structs

Classes are structs with **encapsulation**

Example: PQueue

private fields, public methods that affect fields

Implement methods in .cpp file

```
class Course {
  public:
    void addStudent();
    string getInstructor();
  private:
    int numStudents;
    string professor;
    string name;
};

Course cs106b;
cs106b.addStudent();
```

Sorting

Sorting: Insertion Sort

Works by inserting one element at a time into the sorted list

Sorted list starts as a list of size 1 with the first element in the list, then for all the other elements in the list, we insert each into its proper place in the sorted list

Runtime: O(N^2)

Sorting: Selection Sort

Find the smallest element in the list, and swap it with the leftmost element in the list

Continue by looking at the remaining (unsorted elements)

Runtime: O(N^2)

Sorting: Merge Sort

Split the array into two smaller arrays

Base case: an array of size 1 is trivially sorted

Recursive step: split into two arrays of size (N/2) that we call mergeSort on, then merge the two sorted arrays together

Runtime: O(N log N)

Sorting: QuickSort

Choose an element as a “pivot element”

For all the other elements in the array, split them to the left of the pivot (if they’re smaller than the pivot) or right (if they’re greater). The pivot is now in the correct spot

Recurse on the two arrays on either side of the pivot

Expected: O(N log N)
Worst case: (e.g. picking the smallest element each time) O(N^2)

Pointer Traces

Pointers

You can think of pointers like an address
Building *postOffice = new Building();

The **value** of postOffice is an address, like 531 Lasuen Mall (if you cout postOffice you would get 531 Lasuen Mall).

If you **dereference** or travel to the address specified by postOffice, you get to the actual post office

Pointers - operators to know

Always declared with a * to the right of the type it stores
Read the pointer right to left:
`int **ptr` - ptr is a **pointer** to a **pointer** to an int

Operators:
`*postOffice` - dereferences ptr; goes to the object it stores (travel to the postOffice)
`&postOffice` - gets the address of a variable (doesn't need to be a pointer). The type of the expression is always a pointer to the type of the variable (since `postOffice` is a `Building *`, this is `Building **`)

Pointers and Arrays

When you declare an array, that is also a pointer to the first (0th) element in the array

```
int arr[2];
int *pointer = arr; // pointer points to first
element of arr
//the next lines are equivalent
arr[1] = 3;
pointer[1] = 3; // only works because pointer
secretly points to an array; if it was just one
int, would crash
```

Pointer trace example

```
struct Quidditch {
  int quaffle;
  int *snitch;
  int bludger[2];
};


struct Hogwarts {
  int wizard;
  Quidditch harry;
  Quidditch *potter;
};
```

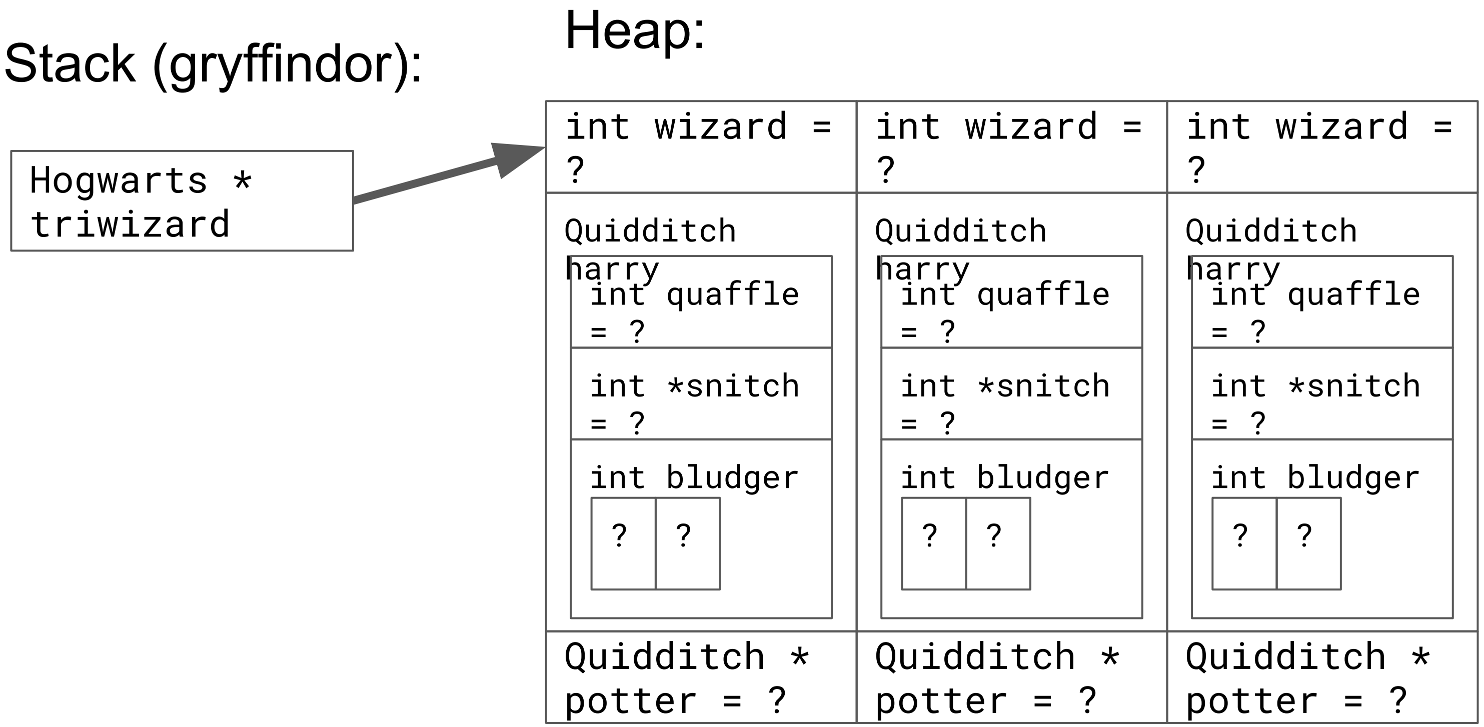
Pointer trace example

```
Quidditch * hufflepuff(Hogwarts * cedric) {
  Quidditch *seeker = &(cedric->harry);
  seeker->snitch = new int;
  *(seeker->snitch) = 2;
  cedric = new Hogwarts;
  cedric->harry.quaffle = 6;
  cedric->potter = seeker;
  cedric->potter->quaffle = 8;
  cedric->potter->snitch =
     &(cedric->potter->bludger[1]);
  seeker->bludger[0] = 4;
  return seeker;
}
```

```
void gryffindor() {
Hogwarts *triwizard = new
Hogwarts[3];
triwizard[1].wizard = 3;
triwizard[1].potter = NULL;
triwizard[0] = triwizard[1];
triwizard[2].potter =
  hufflepuff(triwizard);
triwizard[2].potter->quaffle = 4;
}
```

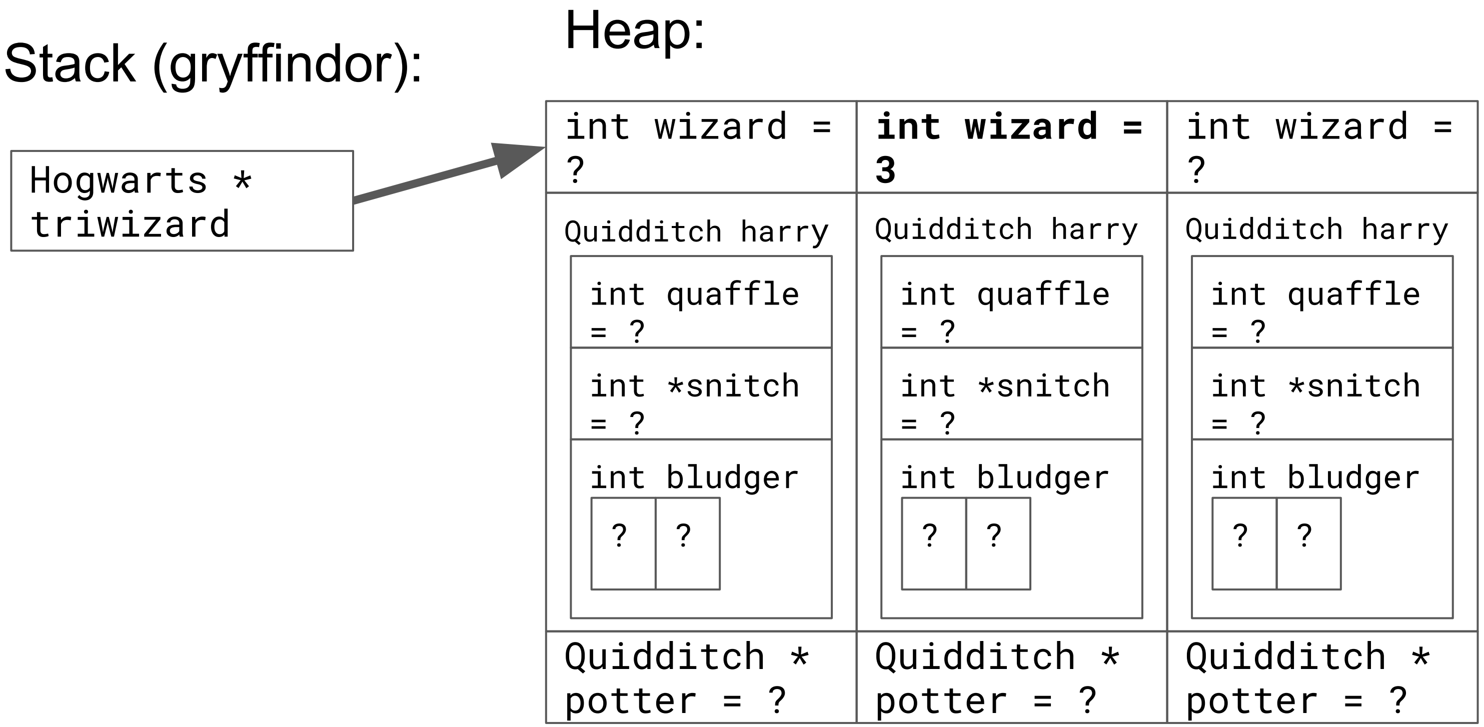
Pointer trace example

```
Hogwarts *triwizard = new Hogwarts[3];
```



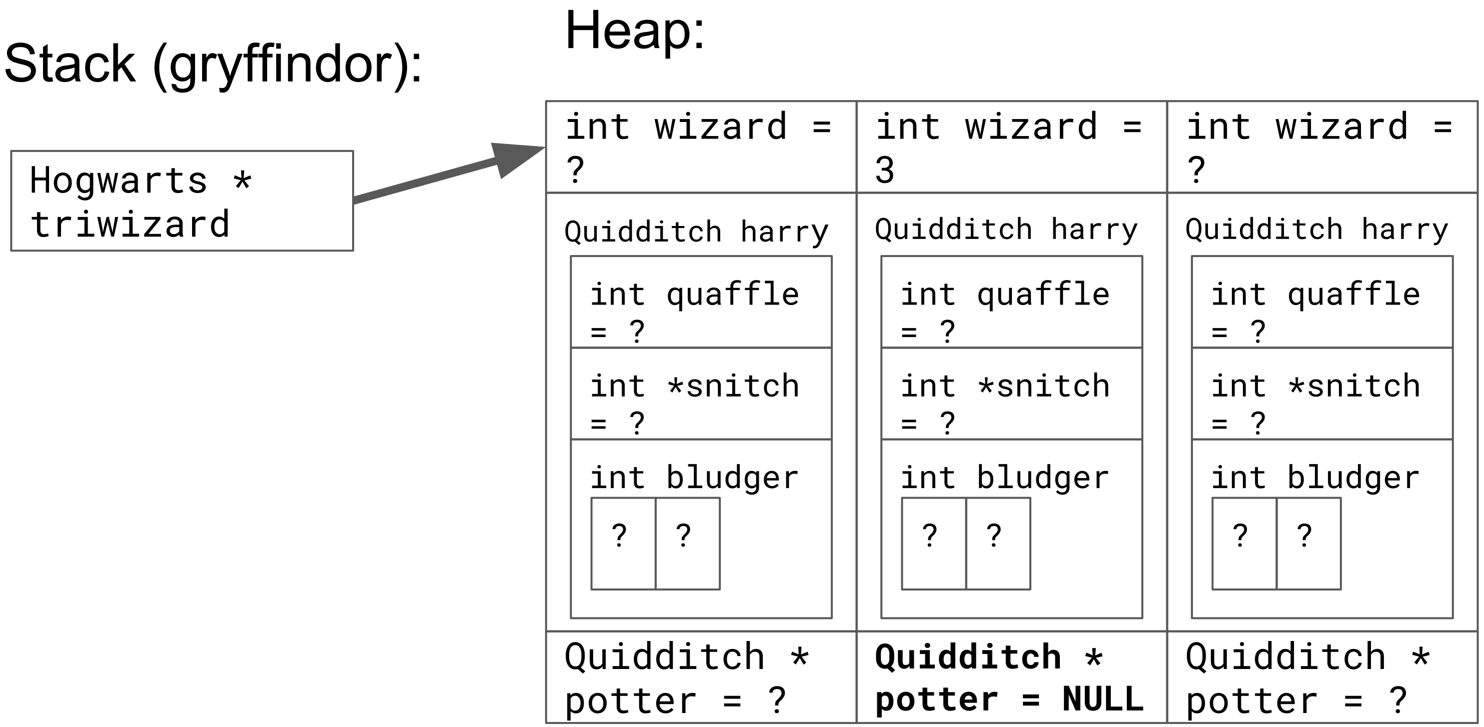
Pointer trace example

```
triwizard[1].wizard = 3;
```



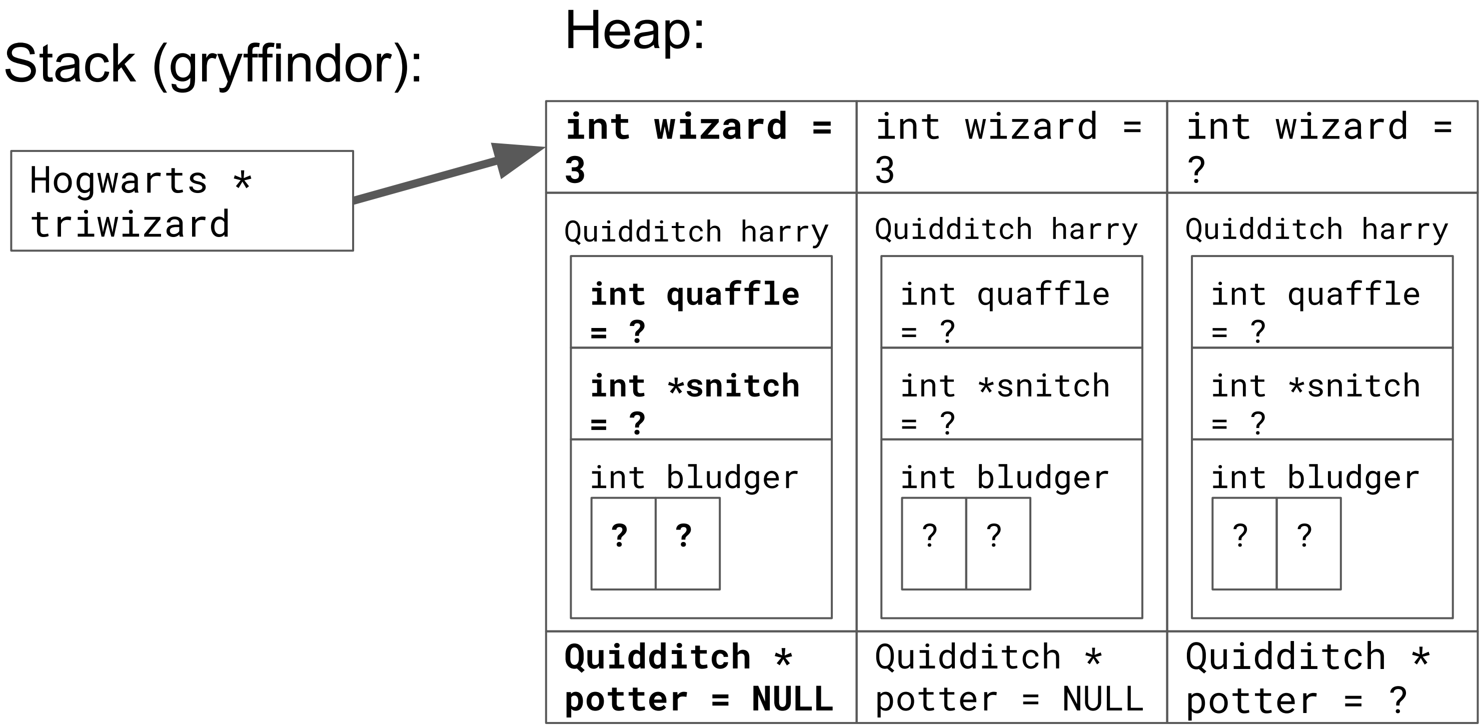
Pointer trace example

```
triwizard[1].potter = NULL;
```



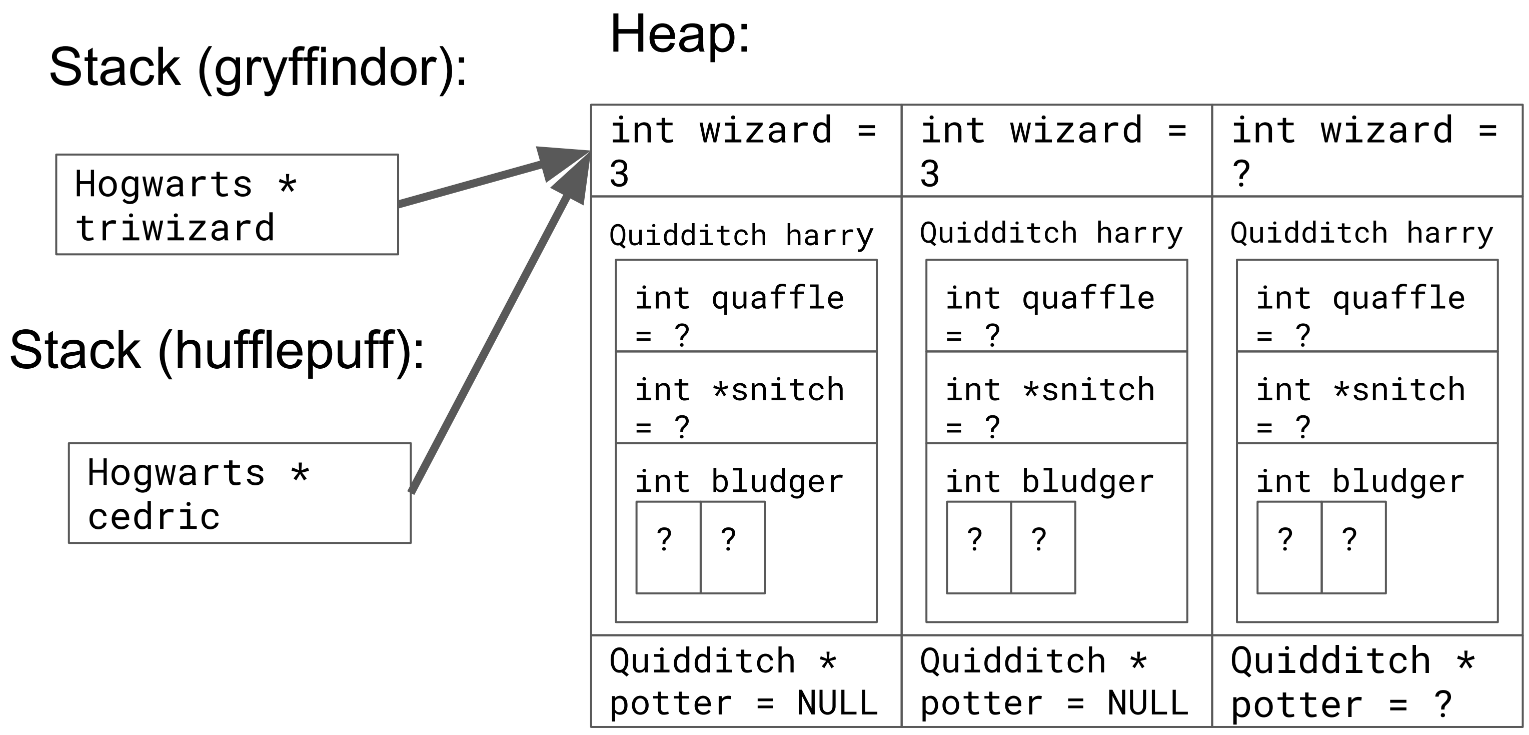
Pointer trace example

```
triwizard[0] = triwizard[1];
```



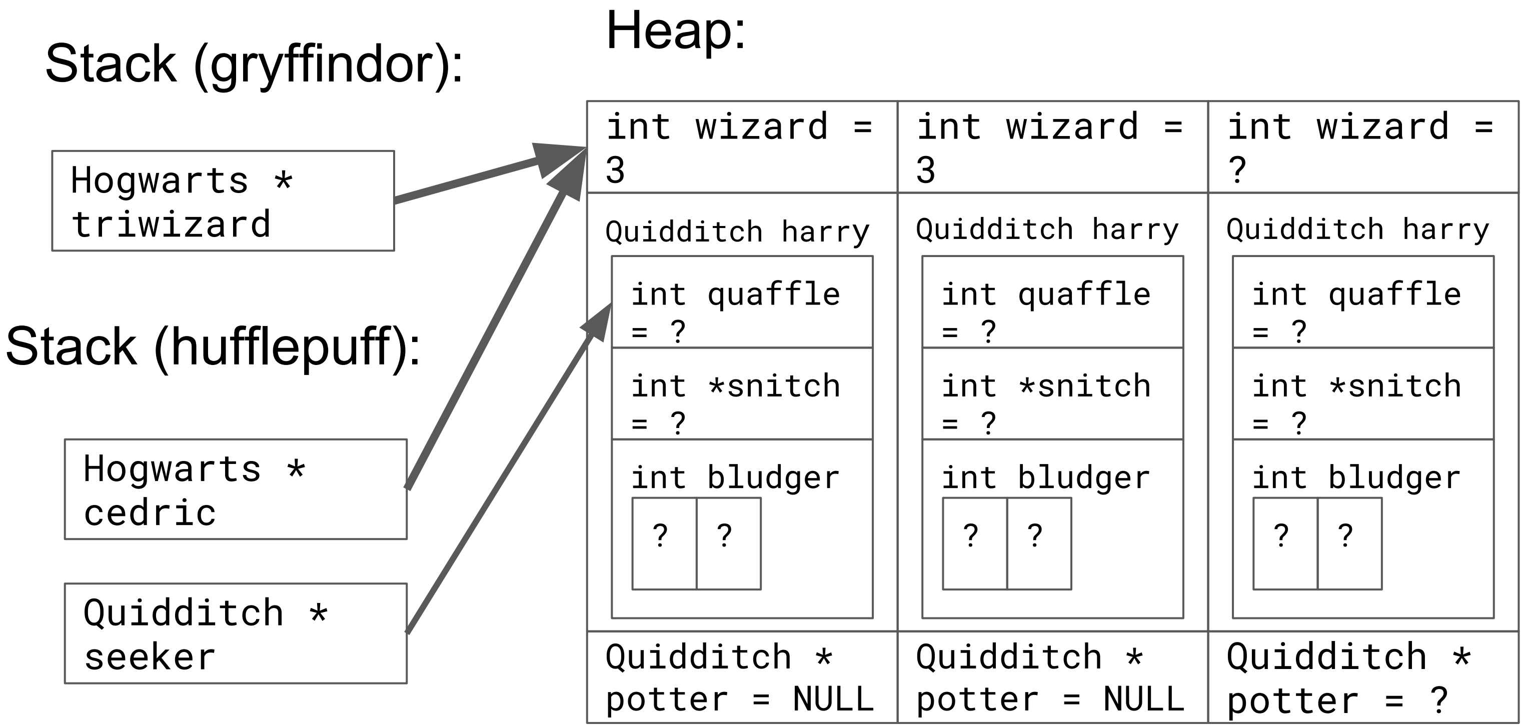
Pointer trace example

```
triwizard[2].potter = hufflepuff(triwizard);
```



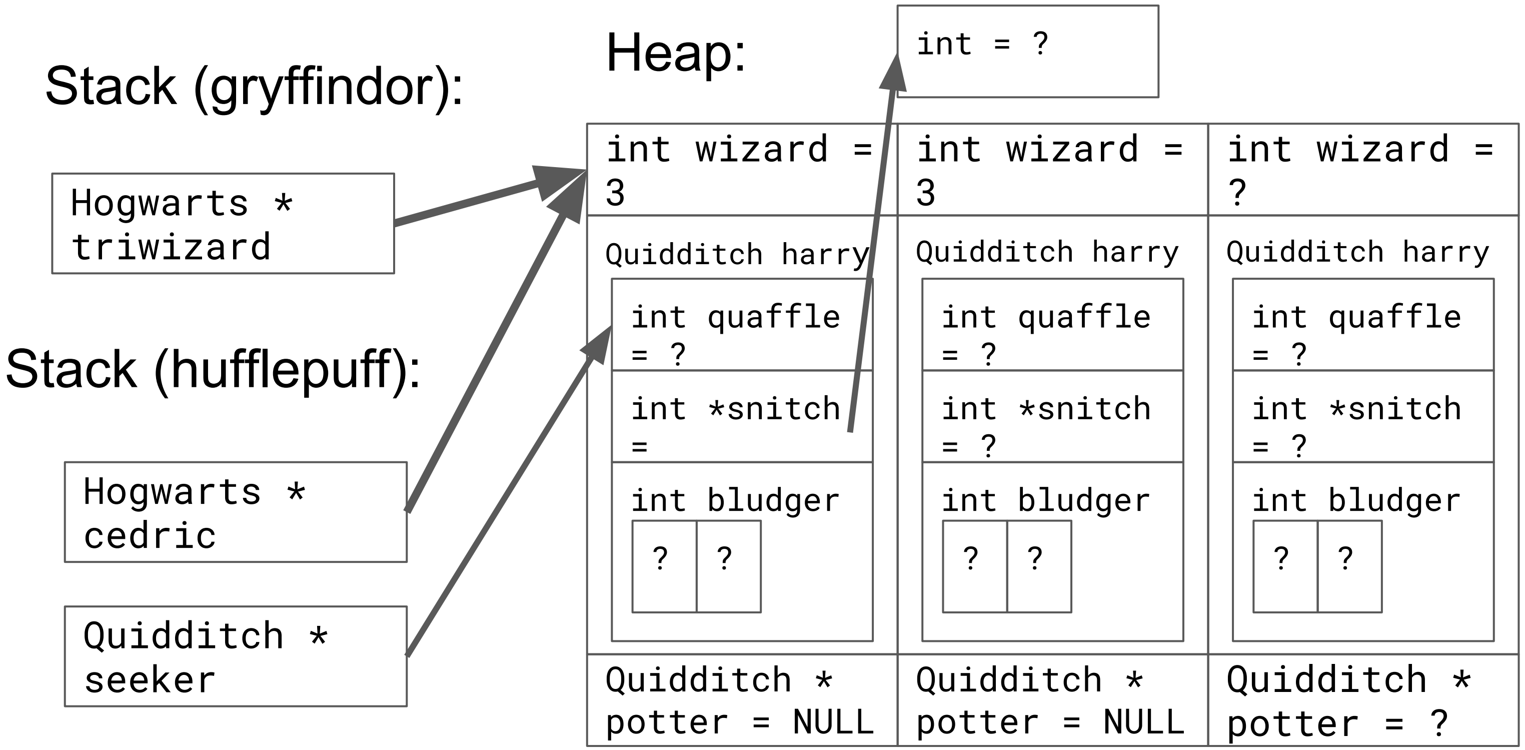
Pointer trace example

```
Quidditch *seeker = &(cedric->harry);
```



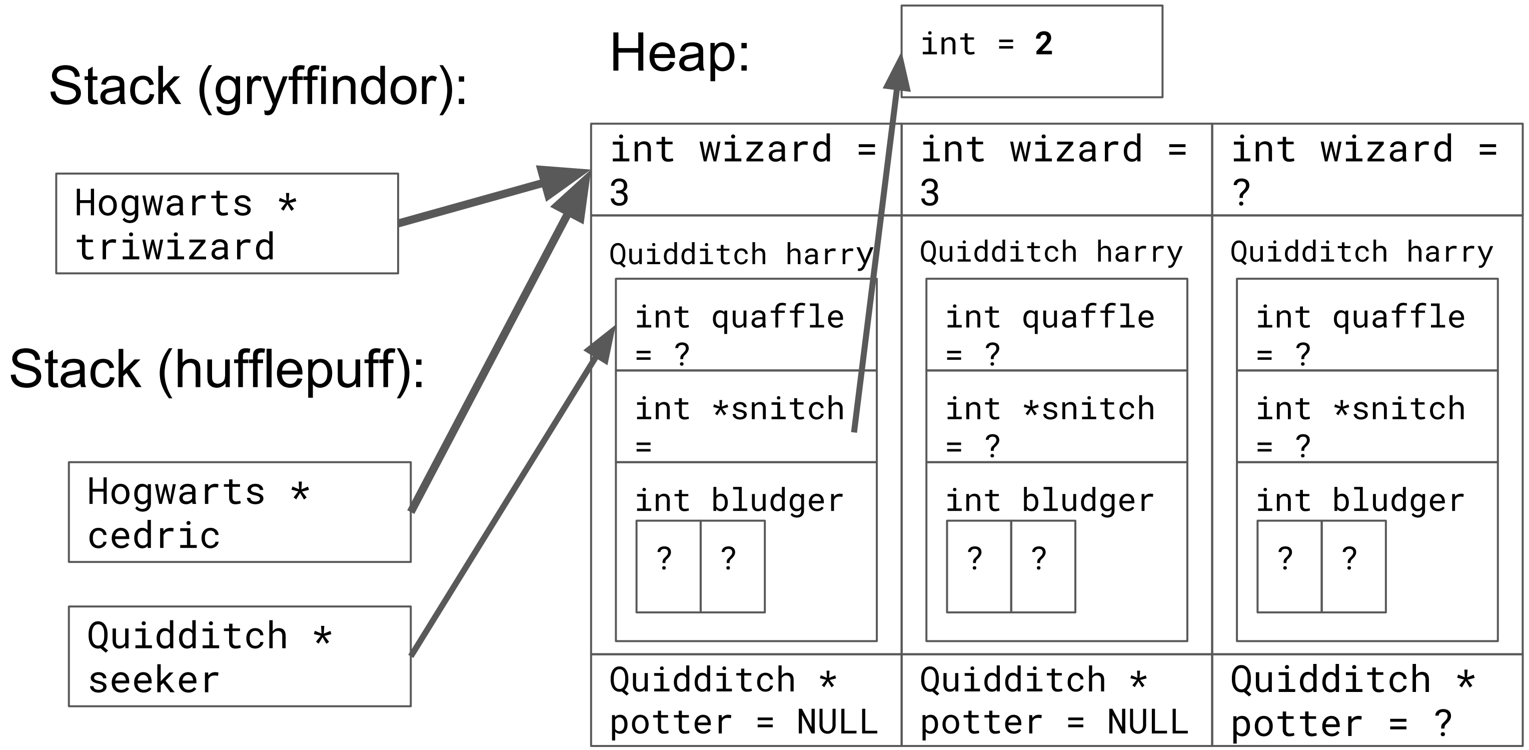
Pointer trace example

```
seeker->snitch = new int;
```



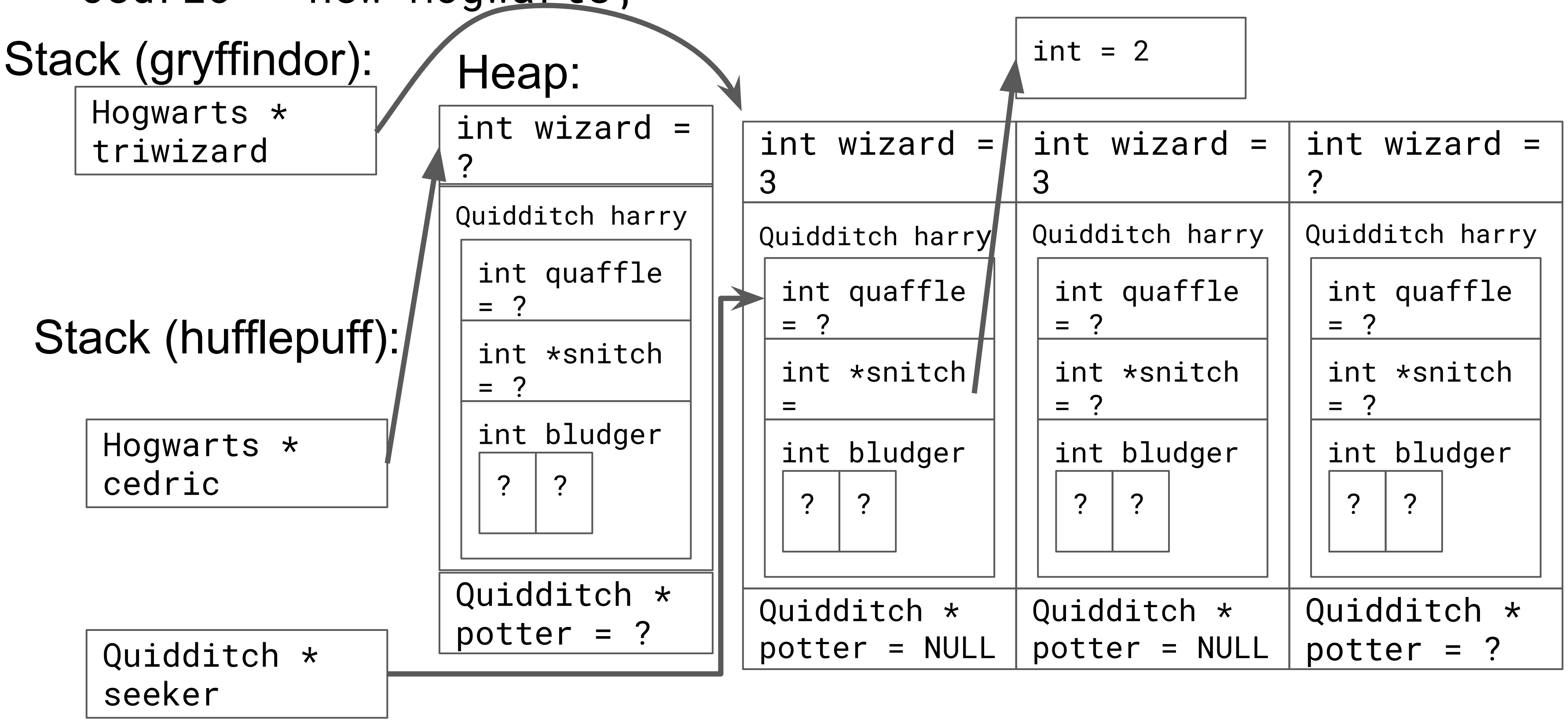
Pointer trace example

```
*(seeker->snitch) = 2;
```



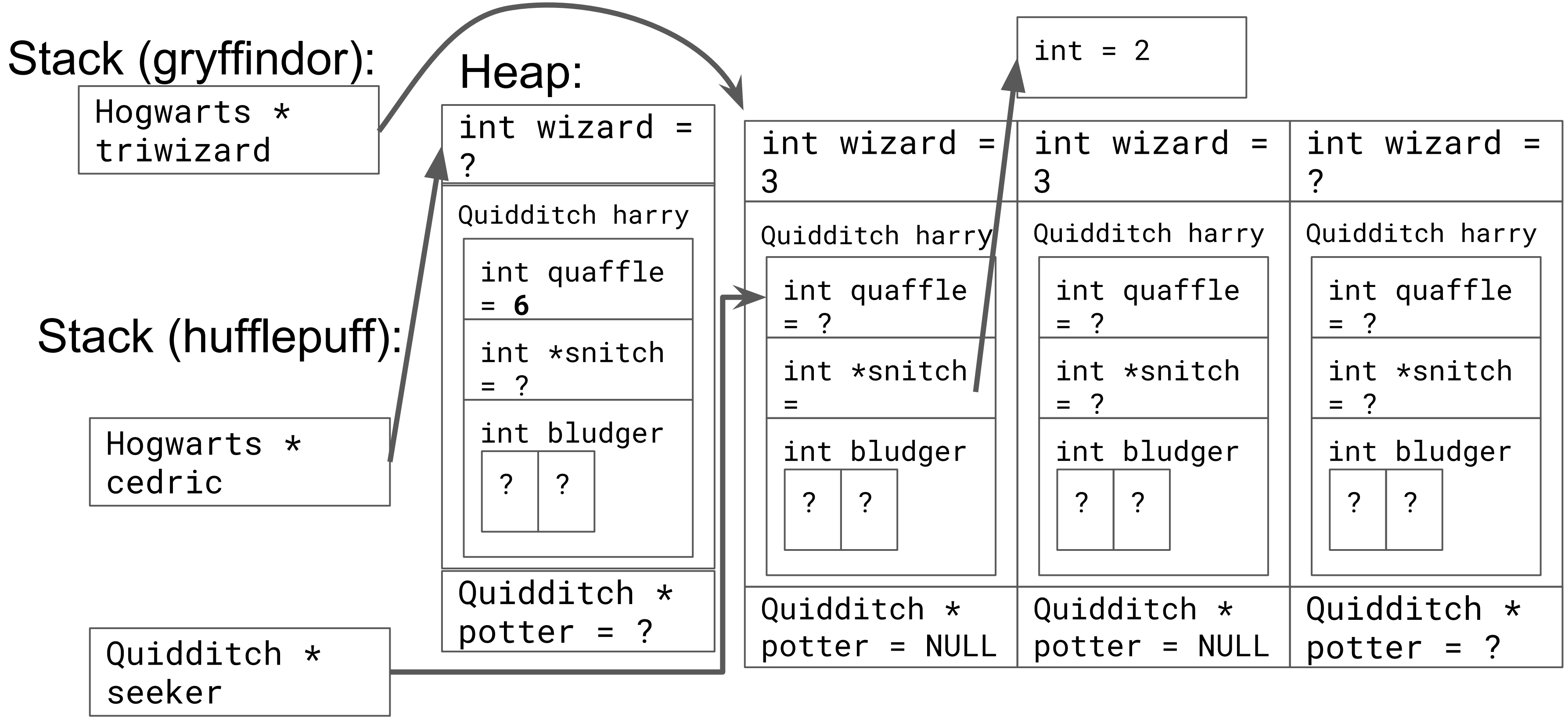
Pointer trace example

```
cedric = new Hogwarts;
```



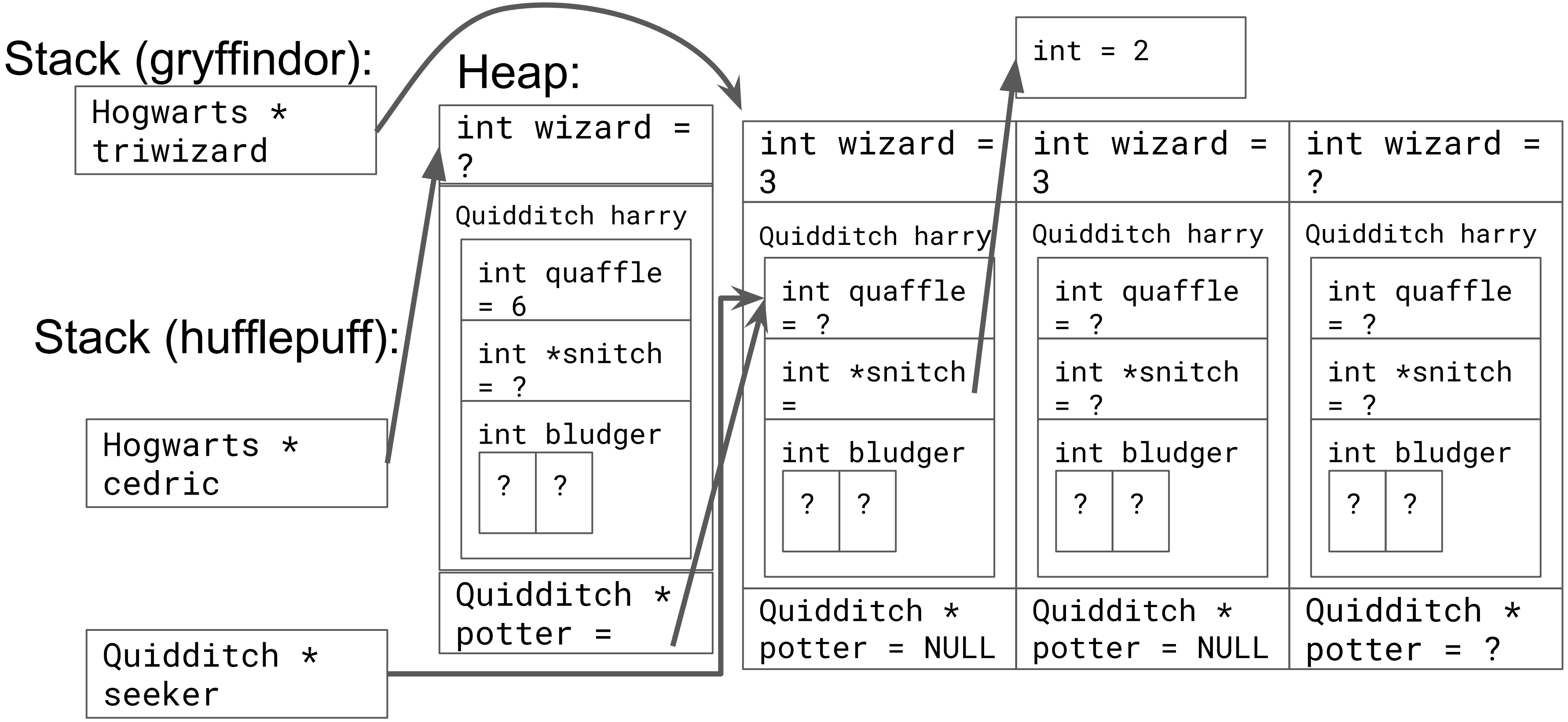
Pointer trace example

```
cedric->harry.quaffle = 6;
```



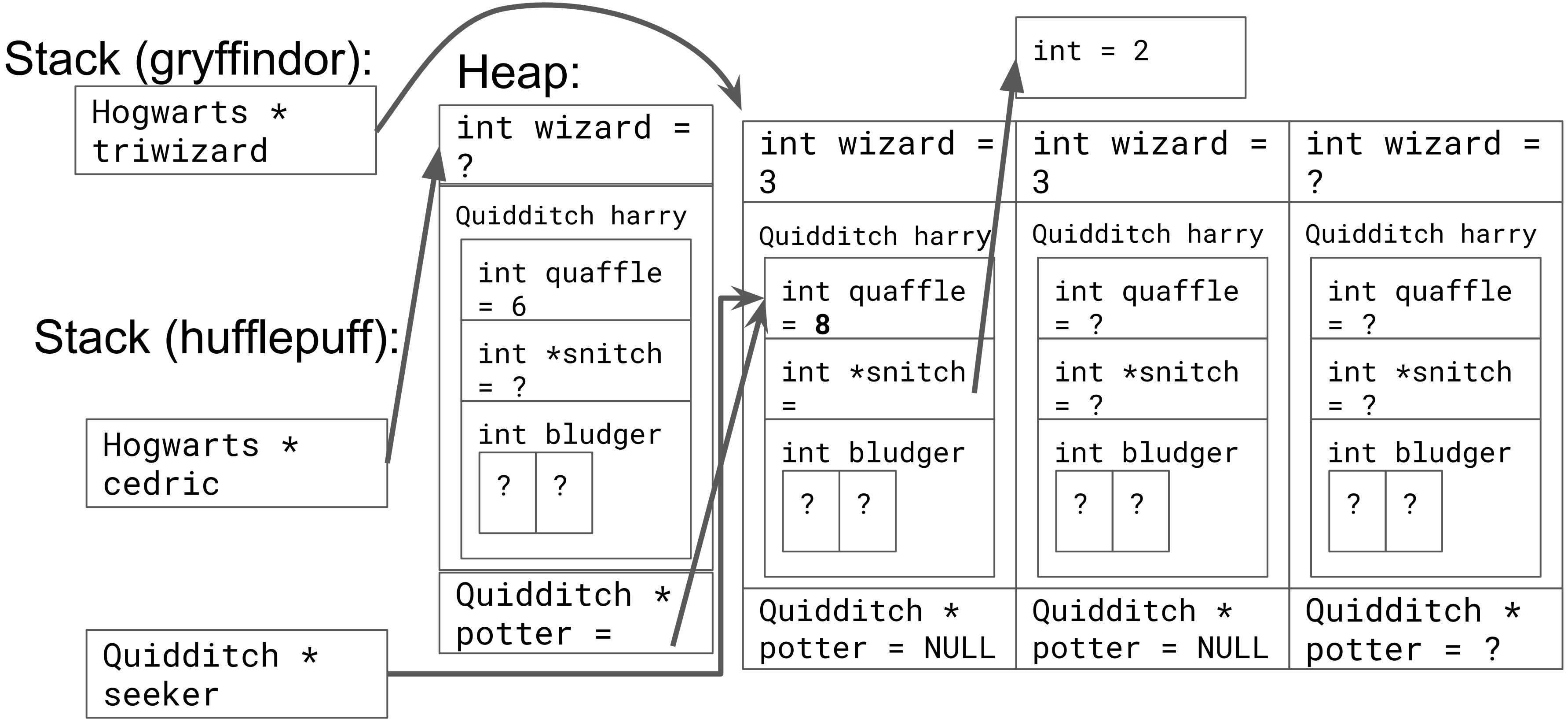
Pointer trace example

```
cedric->potter = seeker;
```



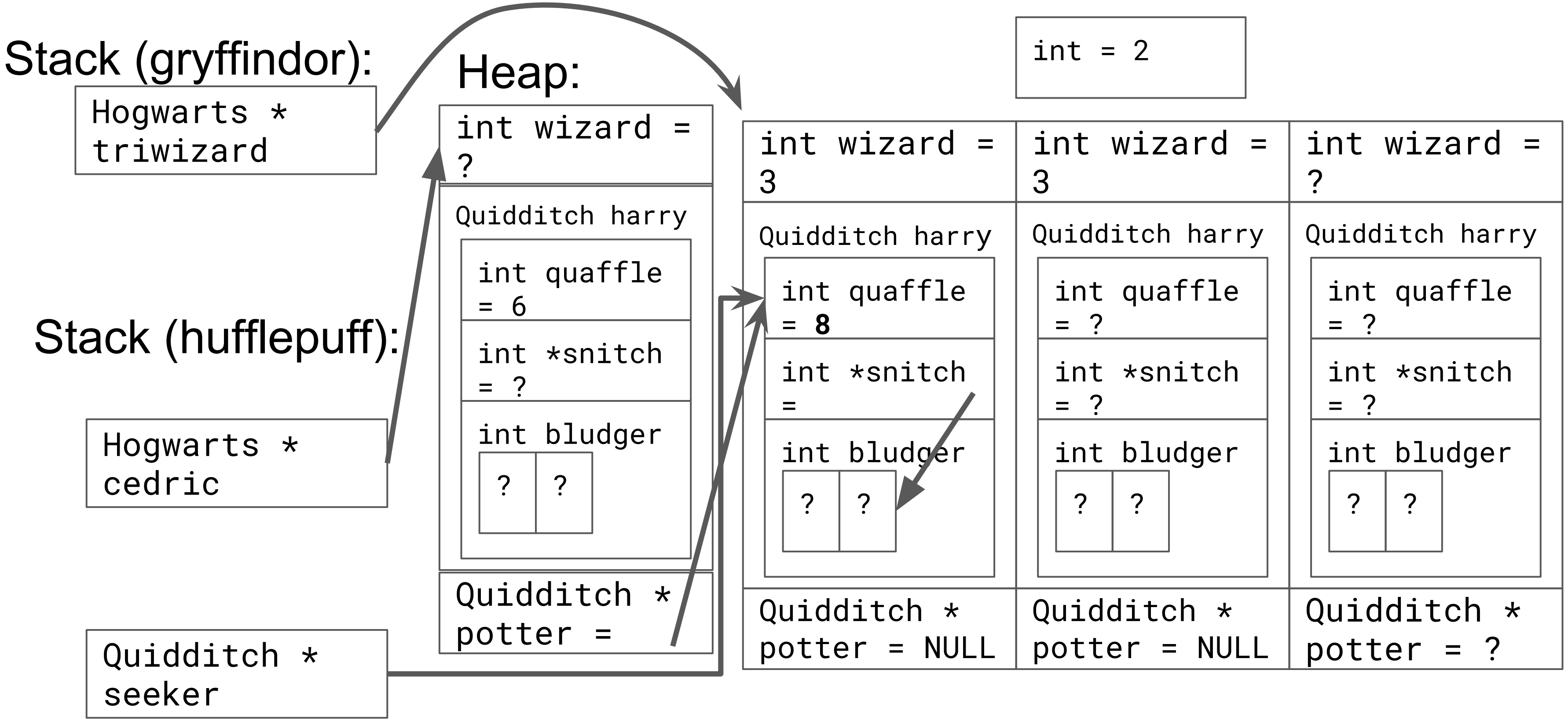
Pointer trace example

```
cedric->potter->quaffle = 8;
```



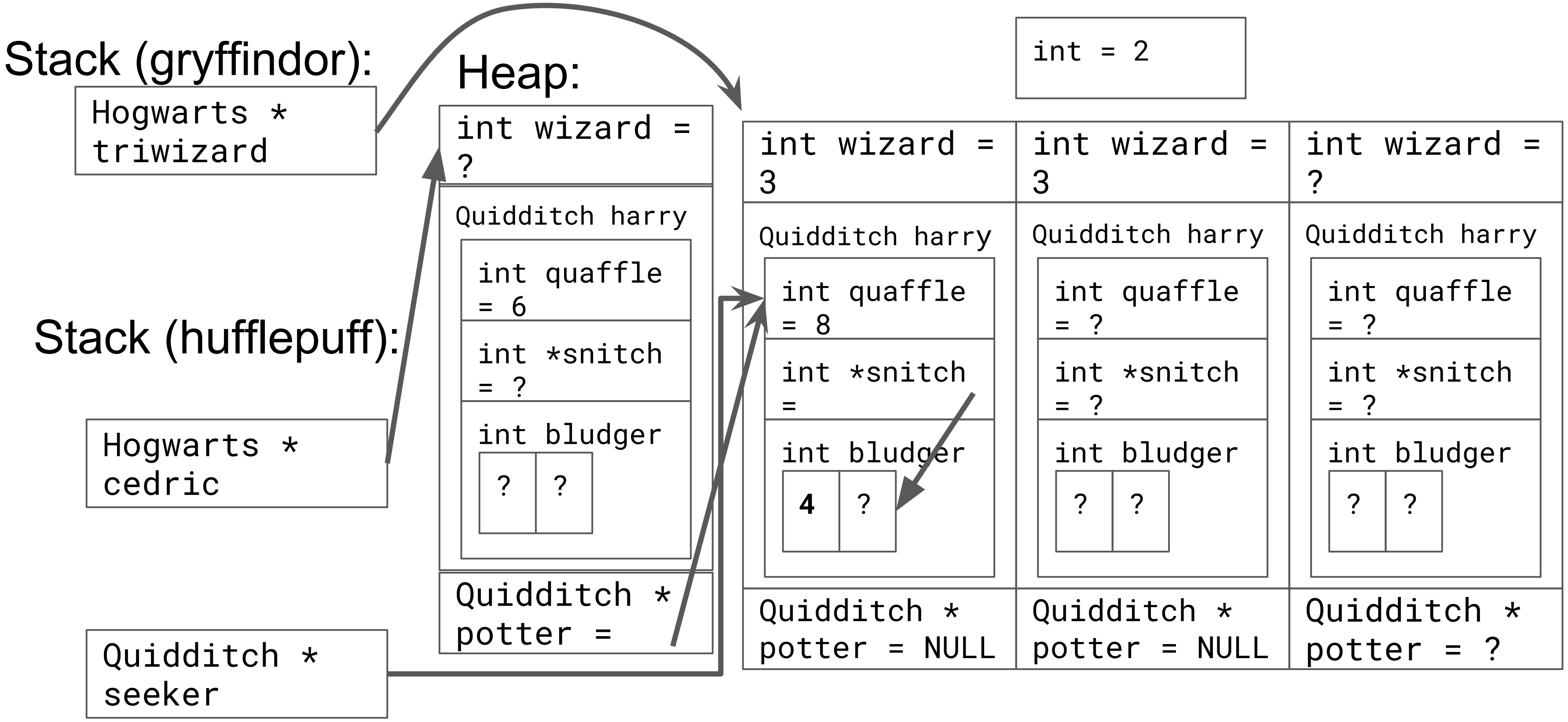
Pointer trace example

```
cedric->potter->snitch = &(cedric->potter->bludger[1]);
```



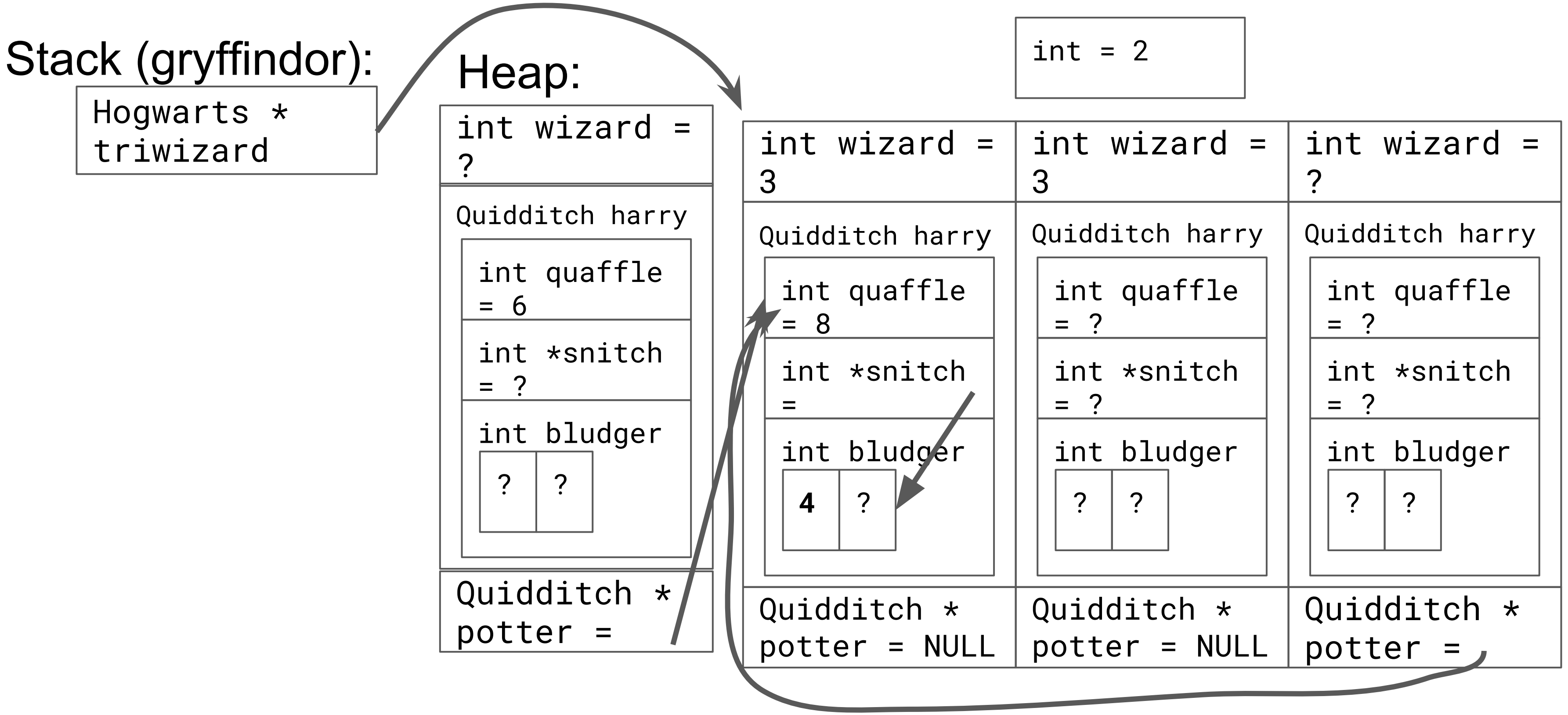
Pointer trace example

```
seeker->bludger[0] = 4;
```



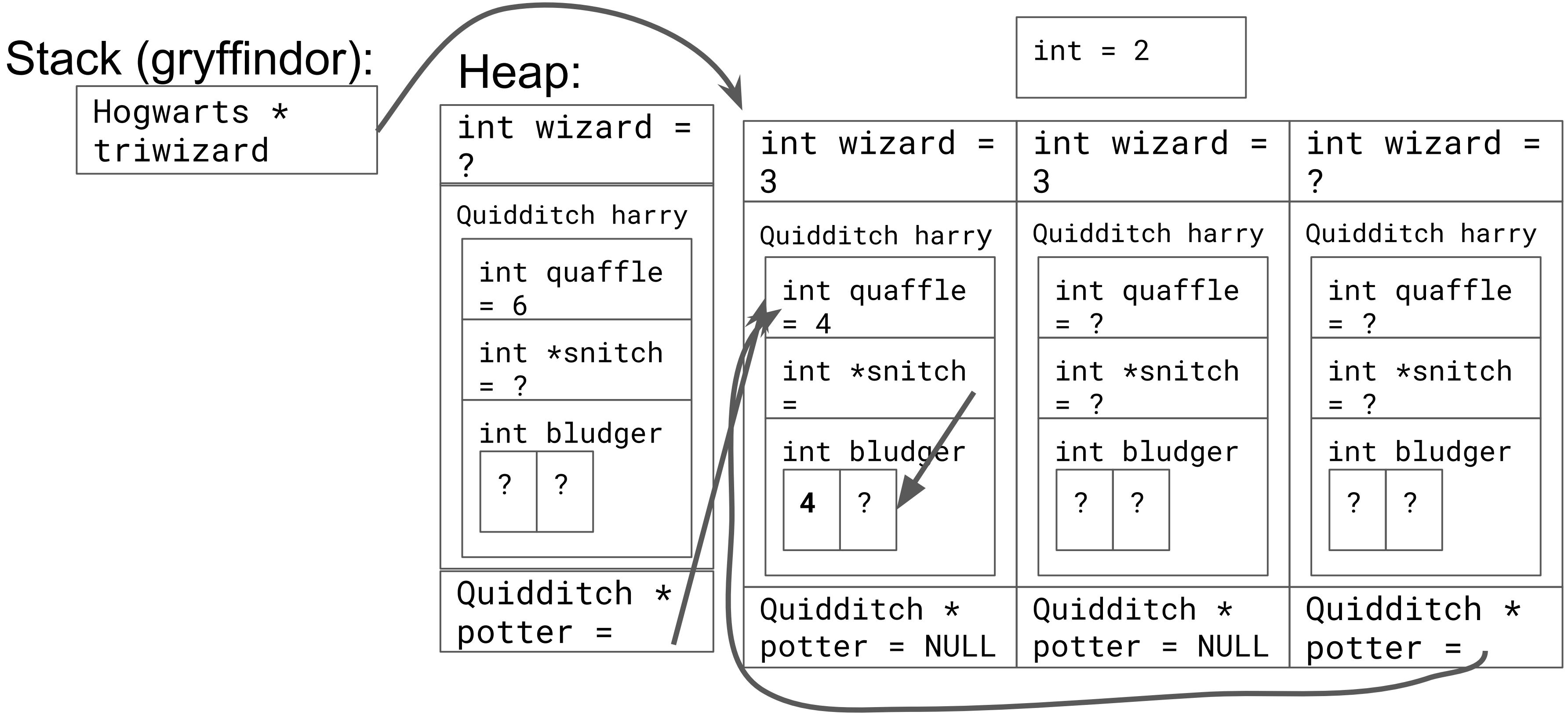
Pointer trace example

```
return seeker / triwizard[2].potter = hufflepuff(triwizard);
```



Pointer trace example

```
triwizard[2].potter->quaffle = 4;
```



Memoization

Memoization

Memoization **caches** or saves information that you’ve already calculated to make your code run more efficiently

Usually used with **recursion**

Memoization - Example

Let’s say you have a chocolate bar that is 1 unit wide and n units long. How many ways can we break this chocolate bar if we can make pieces 1, 2, or 3 pieces long?

Assume that an empty bar can be broken in 1 way

Naive Solution - Chocolate (without Memoization)

```
int numWaysToBreak(int length) {
  if (length < 0) {
    return 0;
  } else if (length == 0) {
    return 1;
  }
  return numWaysToBreak(length - 1) +
         numWaysToBreak(length - 2) +
         numWaysToBreak(length - 3);
}
```

Why Memoization

Let’s say we’re trying to break a chocolate bar of size 4. We end up calculating numWaysToBreak(2) twice (once as part of the call for numWaysToBreak(3) and once as numWaysToCall(4)).

We calculate numWaysToBreak(1) FOUR times (once as part of numWaysToBreak(1), once as numWaysToBreak(2), and twice as numWaysToBreak(3)). A lot of wasted work...

Chocolate With Memoization

```
// create a helper with the cache
int numWaysToBreak(int length) {
  Map<int, int> cache;
  return numWaysToBreak(length, cache);
}
```

Chocolate with Memoization

```
int numWaysToBreak(int length, Map<int, int> & cache) {
  if (length < 0) {
    return 0;
  } else if (length == 0) {
    return 1;
  } else if (cache.containsKey(length)) {
    return cache[length];
  }
  int result = numWaysToBreak(length - 1) +
               numWaysToBreak(length - 2) +
               numWaysToBreak(length - 3);
  cache[length] = result;
  return result;
}
```

LinkedLists

LinkedList Tips

- Draw lots of pictures! Make sure you know exactly where you want things to point, and draw out every step (you want to always have a pointer to everything you want to access)
- Make sure you delete a node when you don’t need it anymore (but after you saved its `next`)
- Good test cases for your list: empty list, list of size 1, list of size 2, list of size 3; try adding/deleting from the beginning, middle, and end

Doubly-Linked List

Previously, we’ve been working with a singly-linked list, where each node has a single next pointer.

Now we’re going to implement a doubly-linked list where each node has a next and a prev pointer: (we assume a constructor of DoubleNode(value, prev, next))

```
struct DoubleNode {
  string value;
  DoubleNode *next;
  DoubleNode *prev;
};
```

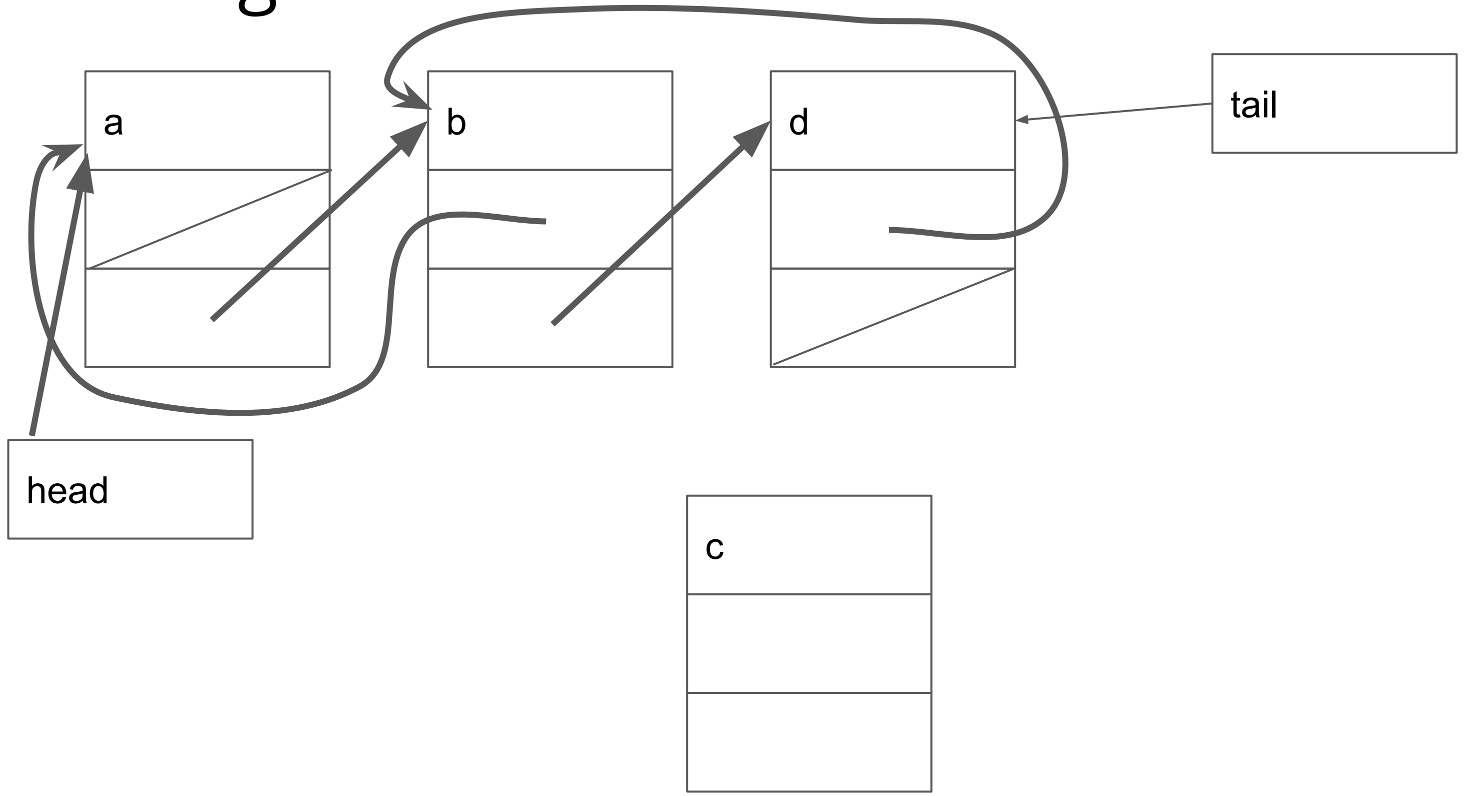
Our Sorted DoublyLinkedList class

```
class DoublyLinkedList {
  public:
    void add(String & elem);
    void delete(String & elem);
    int findPosition(String & elem);
  private:
    DoubleNode *head;
    DoubleNode *tail;
};
```

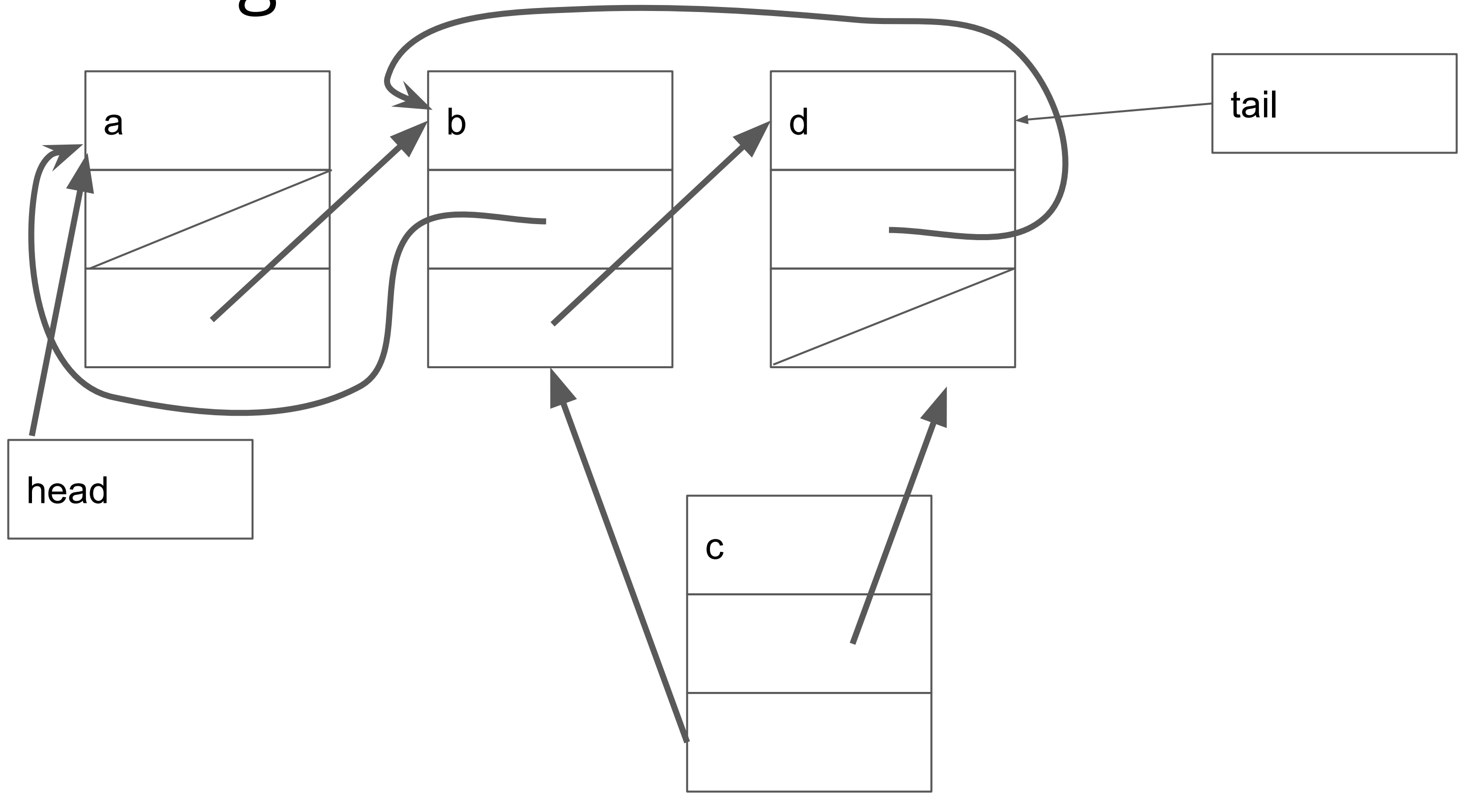
Adding to our sorted doubly linked list

- Empty list: head = tail = new node
- New node is less than head: head will be new node; new node’s next will point to head; old head’s prev = new head
- New node is greater than tail: tail will be new node; new node’s previous will point to old tail; old tail’s next points to new tail
- Inserting into the middle of the list (see next slide)

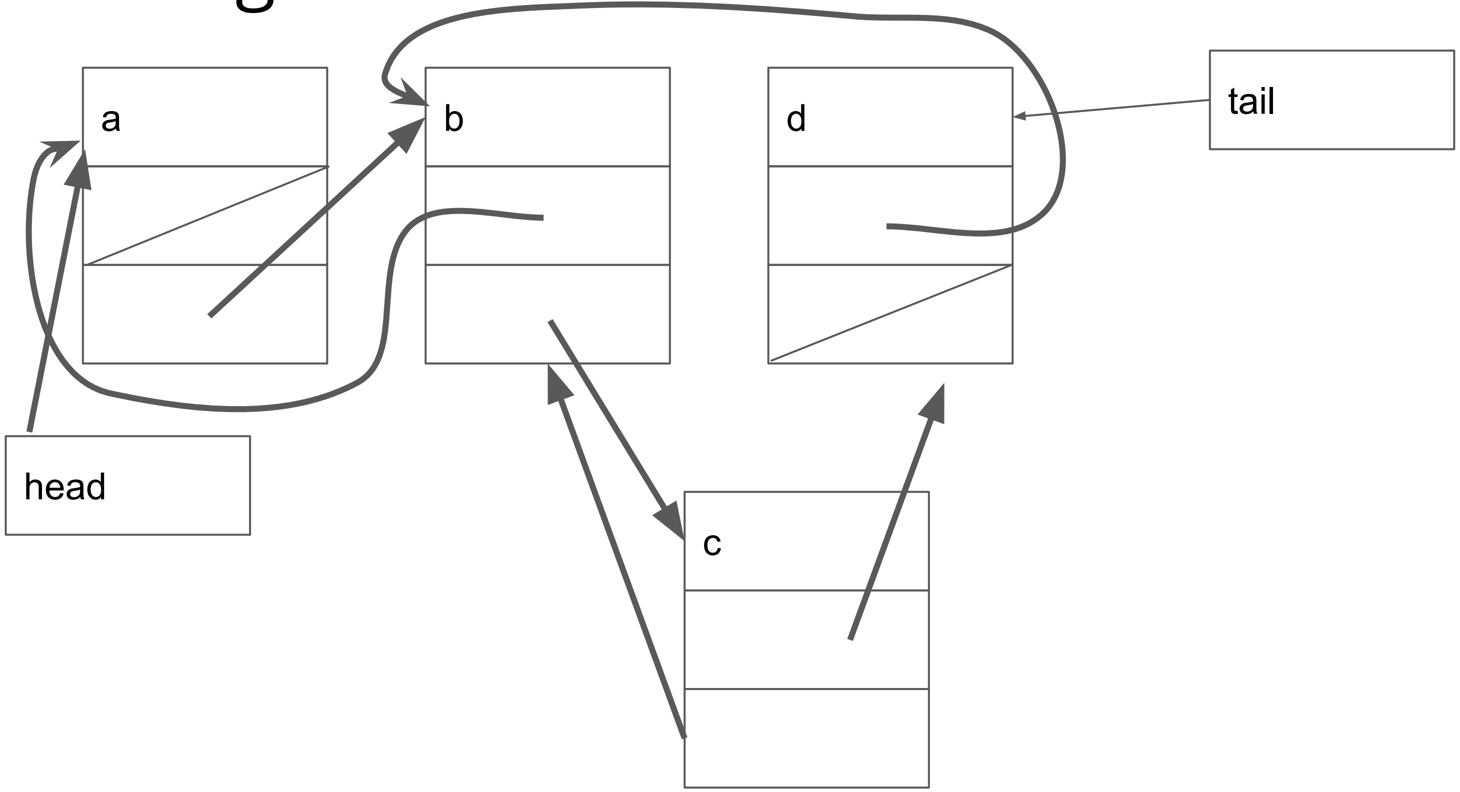
Inserting into the middle of the list



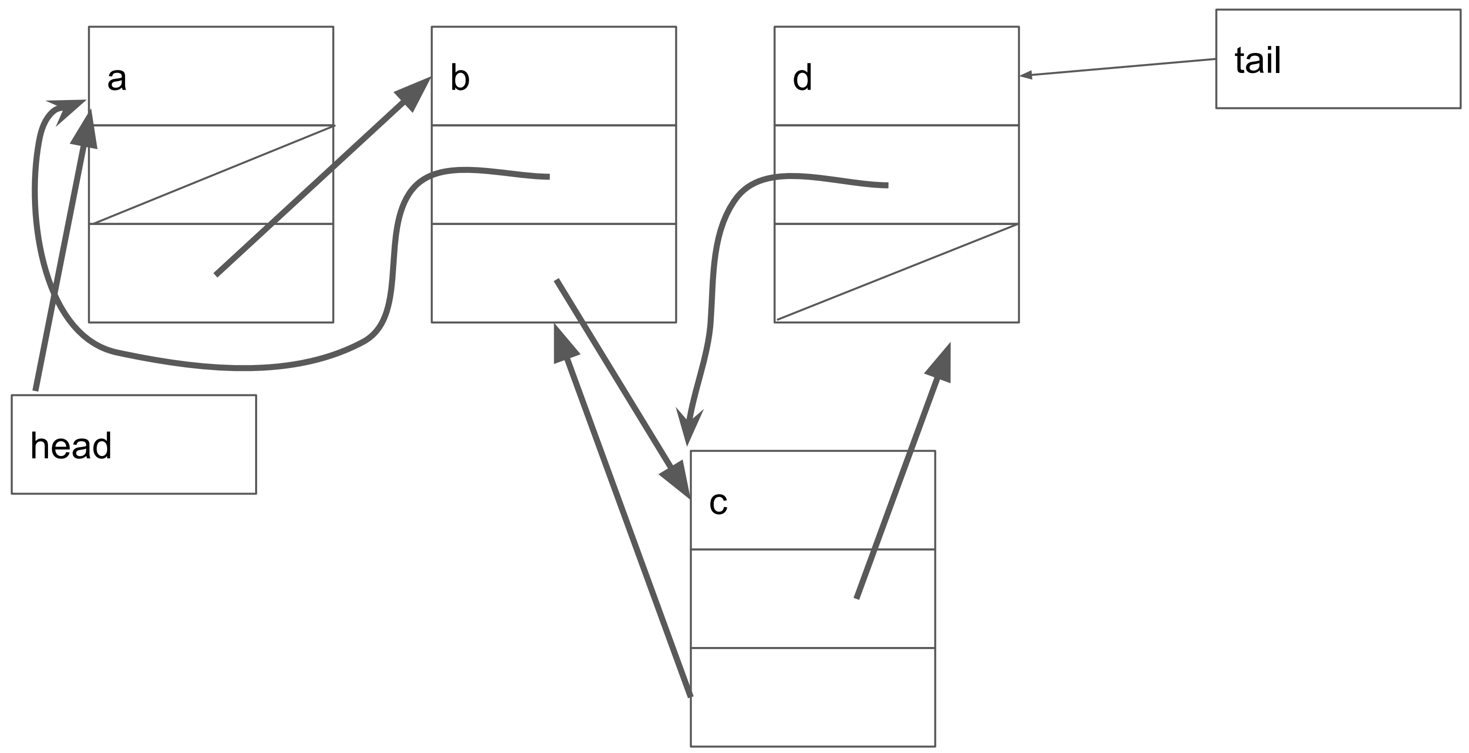
Inserting into the middle of the list



Inserting into the middle of the list



Inserting into the middle of the list



Adding to our sorted doubly linked list

```
void DoublyLinkedList::add(String &elem) {
  // nothing in the list or smallest value in list
  if (head == null || head->value > elem->value) {
    if (head != NULL) {
      head->prev = new DoubleNode(elem, NULL, head);
      head = head->prev;
    } else {
      head = tail = new DoubleNode(elem, NULL, NULL);
    }
  } else if (tail->value <= elem->value) {
    // elem is greatest element in list
    tail->next = new DoubleNode(elem, tail, NULL);
    tail = tail->next;
  } // finding the correct spot is on the next page
```

Adding to our sorted doubly linked list

```
void DoublyLinkedList::add(String &elem) {
  // continued from previous page
  else {
    DoubleNode *curr = head;
    while (curr->value < elem) {
      curr = curr->next;
    }
    DoubleNode *toInsert = new DoubleNode(elem, curr->prev, curr);
    DoubleNode *prev = curr->prev;
    curr->prev = toInsert;
    //make sure that the node before inserted points to inserted
    prev->next = toInsert;
  }
}
```

Finding the position of an element

```
void DoublyLinkedList::findPosition(String &elem) {
  int position = 0;
  DoubleNode *curr = head;
  while (curr != NULL) {
    if (curr->value == elem) {
      return position;
    }
    curr = curr->next;
    position++;
  }
  return -1; // not found
}
```

Deleting from our sorted doubly linked list

```
void DoublyLinkedList::delete(String &elem) {
  if (head == NULL) {
    throw str(“This list is empty”);
  }
  if (head->value == elem) {
    DoubleNode *trash = head;
    head = head->next;
    if (head == NULL) {
      tail = NULL;
    } else {
      head->prev = NULL;
    }
    delete trash;
    return;
  }
```

Deleting from our sorted doubly linked list

```
void DoublyLinkedList::delete(String &elem) {
  // look for node to delete
  DoubleNode *curr = head;
  while (curr != tail && curr->value < elem) {
    curr = curr->next;
  }
  if (curr->value != elem) return; // not found
  if (curr == tail) {
    DoubleNode *trash = tail;
    tail = tail->prev;
    tail->next = NULL;
    delete trash;
  }
```

Deleting from our sorted doubly linked list

```
void DoublyLinkedList::delete(String &elem) {
  else { // not deleting the tail
    ListNode *trash = curr;
    curr->prev->next = curr->next; // previous points to next
    curr->next->prev = curr->prev; // next points to previous
    delete trash;
  }
}
```

Extra Practice

- Traverse (i.e. read every element in a LinkedList) without notes (you can use the size function of your PQueue for reference)
- Add an element to a LinkedList (you can use the add function of your PQueue for reference)
- Delete an element from a LinkedList (use changePriority of PQueue for reference)
- Check out the section six handout for lots of practice

Hashing!

Hash Functions

Basic definition: a hash function maps something (like an int or string) to a number

A **valid** hash function will always return the same number given two inputs that are considered equal

A **good** hash function distributes the values uniformly over all the numbers

Hash Functions: Good or Bad

```
struct BankAccount {
  int routingNumber;
  int amount;
};
```

Two bank account objects are considered equal if they have the same routing number.

```
int hash(BankAccount account) {
  return randomInteger(0, 100);
}
```

Hash Functions: Good or Bad

```
struct BankAccount {
  int routingNumber;
  int amount;
};
```

INVALID - given the same bank account, might return any random number

Two bank account objects are considered equal if they have the same routing number.

```
int hash(BankAccount account) {
  return randomInteger(0, 100);
}
```

Hash Functions: Good or Bad

```
struct BankAccount {
  int routingNumber;
  int amount;
};
```

Two bank account objects are considered equal if they have the same routing number.

```
int hash(BankAccount account) {
  return account.routingNumber % 2;
}
```

Hash Functions: Good or Bad

```
struct BankAccount {
  int routingNumber;
  int amount;
};
```

Two bank account objects are considered equal if they have the same routing number.

```
int hash(BankAccount account) {
  return account.routingNumber % 2;
}
```

VALID but Not Good - will generate the same result for two accounts that are considered the same, but not uniformly spread over all the integers

Hash Functions: Good or Bad

```
struct BankAccount {
  int routingNumber;
  int amount;
};
```

Two bank account objects are considered equal if they have the same routing number.

```
int hash(BankAccount account) {
  return account.amount % 100;
}
```

Hash Functions: Good or Bad

```
struct BankAccount {
  int routingNumber;
  int amount;
};
```

INVALID - The bank account amount might change, leading to accounts with the same routing number being put in different buckets

Two bank account objects are considered equal if they have the same routing number.

```
int hash(BankAccount account) {
  return account.amount % 100;
}
```

Hash Functions: Good or Bad

```
struct BankAccount {
  int routingNumber;
  int amount;
};
```

Two bank account objects are considered equal if they have the same routing number.

```
int hash(BankAccount account) {
  long hash = (account.routingNumber *
265443761L) % INT_MAX;
  return hash;
}
```

Hash Functions: Good or Bad

```
struct BankAccount {
  int routingNumber;
  int amount;
};
```

Two bank account objects are considered equal if they have the same routing number.

```
int hash(BankAccount account) {
  return (account.routingNumber * 265443761L) %
INT_MAX;

}
```

VALID and GOOD - given the same routing number, this will always return the same number, and it's spread relatively evenly over all possible (positive) integers

*Taken from StackOverflow

Using Hash Functions

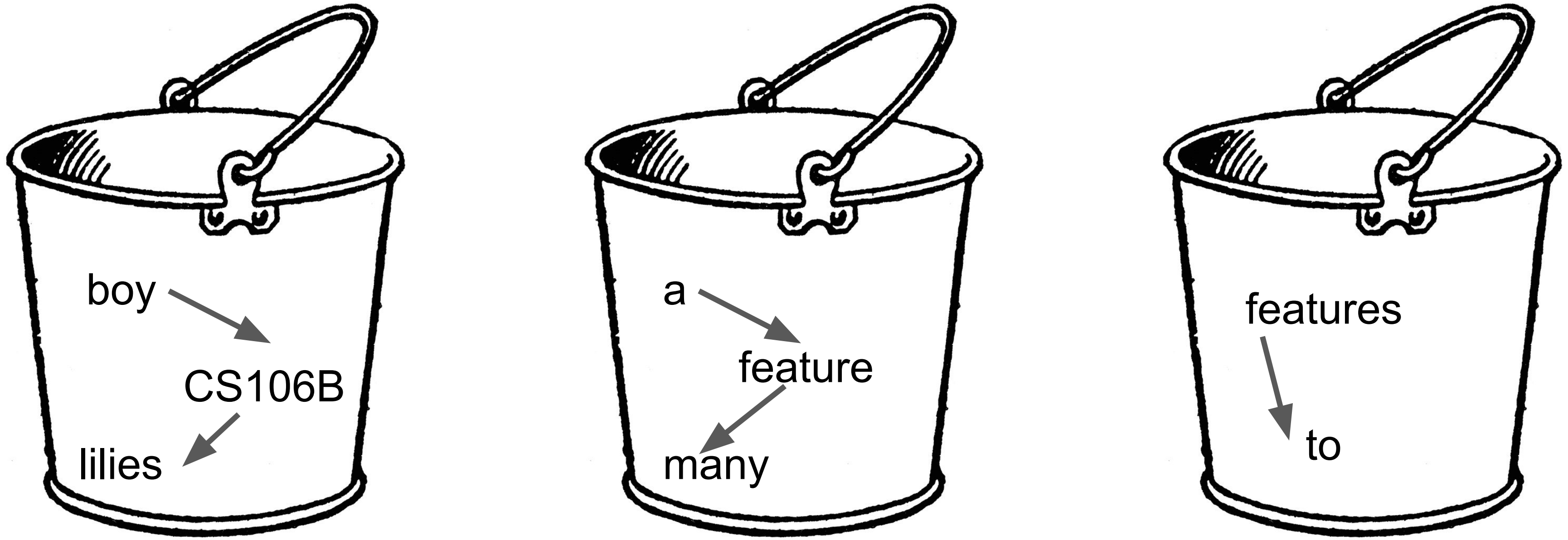
Hash Functions are used to assign elements to buckets for a HashSet or HashMap.

```
int bucket(elem) {
  return hash(elem) % numBuckets;
}
```

Using Hash Functions

Let’s say our hash function returned the length of the string we’re hashing and we have 3 buckets. Our buckets may look like:

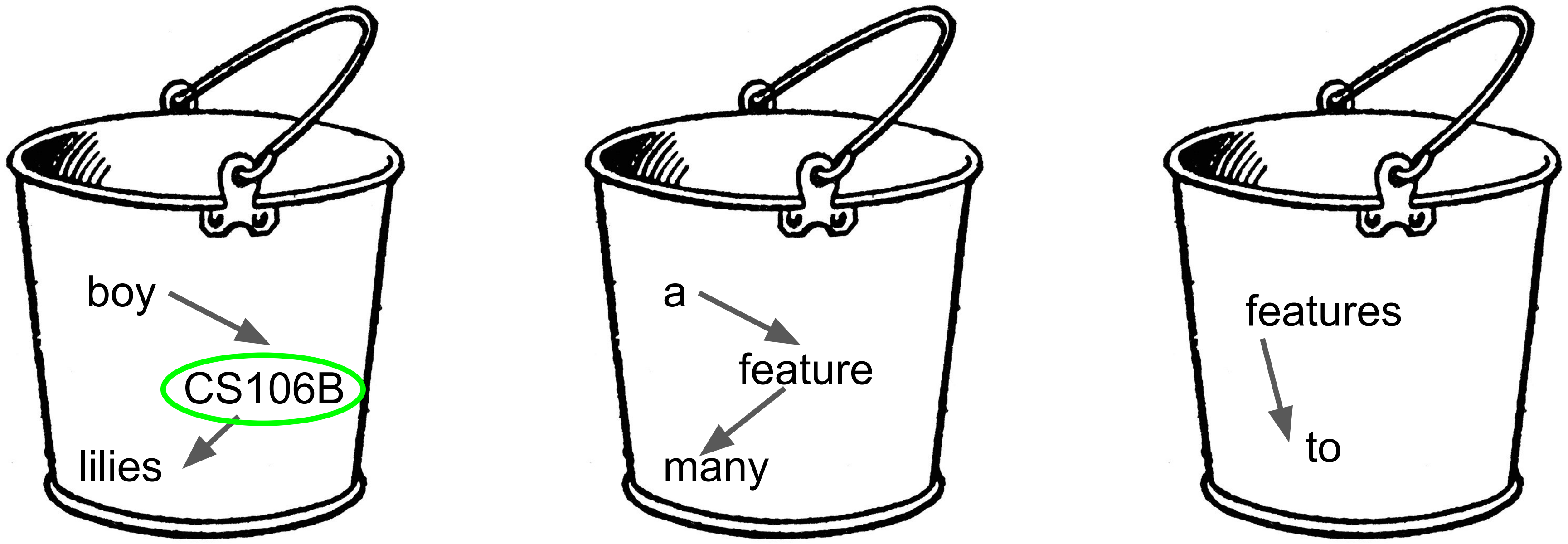
Inside each bucket, we have a LinkedList of elements



Using Hash Functions

When we search for an element, we look in it’s bucket

Search CS106B: look in bucket 0 and look through the whole LinkedList

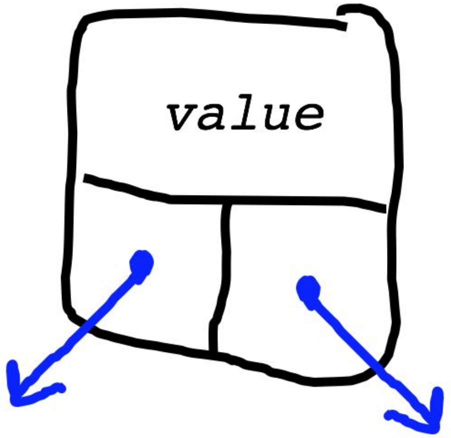


Trees!

Trees!

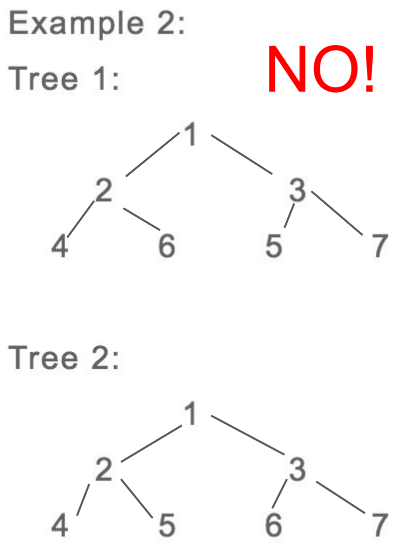
- Just linked lists with > 1 next pointer, no cycles, and max 1 parent per node
- Ex: Binary Trees, Trinary Trees, N-ary Trees

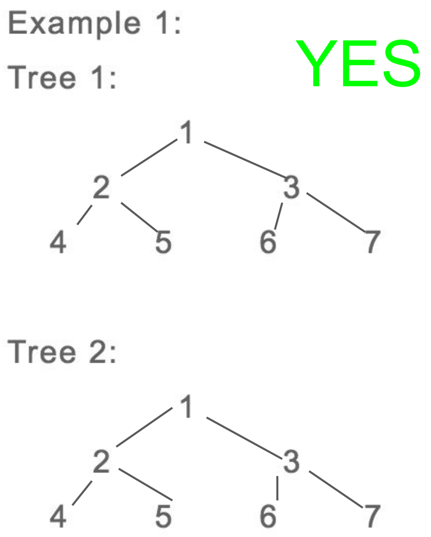
```
struct Tree {
  string value;
  Tree * left;
  Tree * right;
};
```



Tree Exercise I: Traversing

Write a Boolean recursive function `areIdentical(TreeNode * root1, TreeNode* root2)` that checks if two binary trees are identical.





Tree Exercise I - **Recursion!**

```
bool areIdentical(TreeNode *root1, TreeNode *root2) {
   if (root1 == NULL && root2 == NULL) {
      return true;
   } else if (root1 != NULL && root2 != NULL) {
      return areIdentical(root1->left) && areIdentical(root1->right) &&
             areIdentical(root2->left) && areIdentical(root2->right)
   } else {
      return false;
   }
}
```

Program logic:

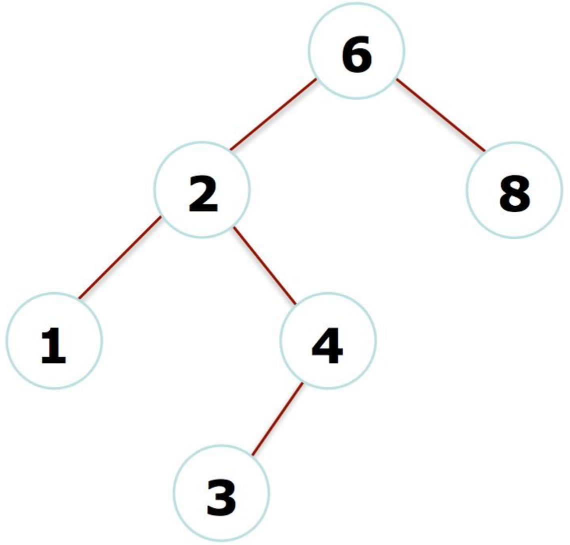
- Check for value equivalence at each node
- Traverse the trees simultaneously

Binary Search Trees!

BSTs

For every node, X, all the items in its left subtree are smaller than X, and the items in the right tree are larger than X.

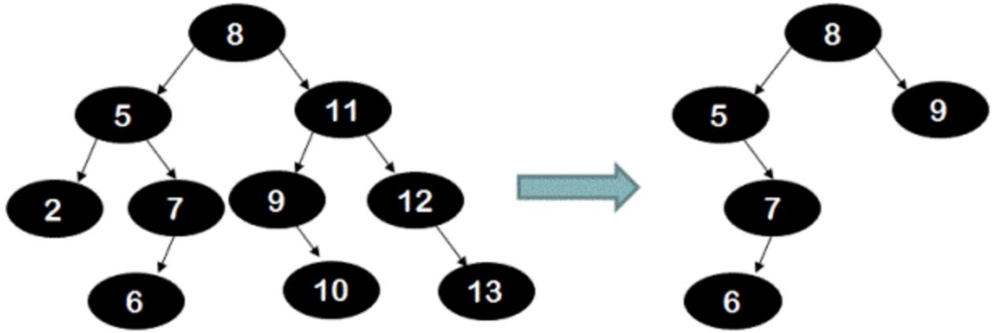
Need to balance



BST Exercise I

Write a `BinaryTree` member function `BinaryTree::keepRange(int min, int max)` that removes all nodes outside of that range (inclusive) and maintains Binary Tree structure.

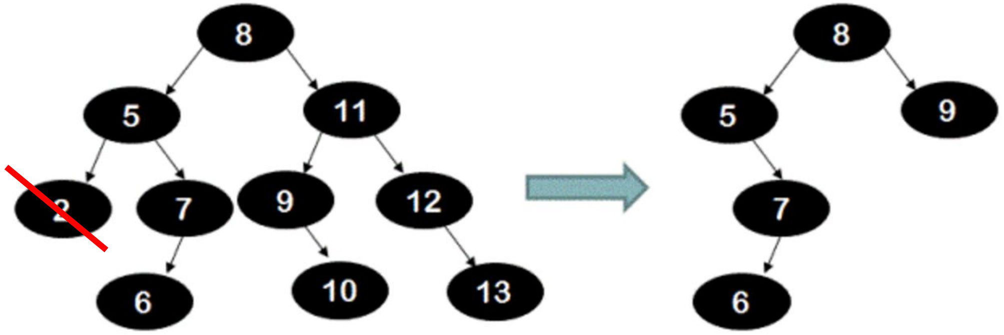
`keepRange(3,9)`



BST Exercise I

Write a BinaryTree member function `BinaryTree::keepRange(int min, int max)` that removes all nodes outside of that range (inclusive) and maintains Binary Tree structure.

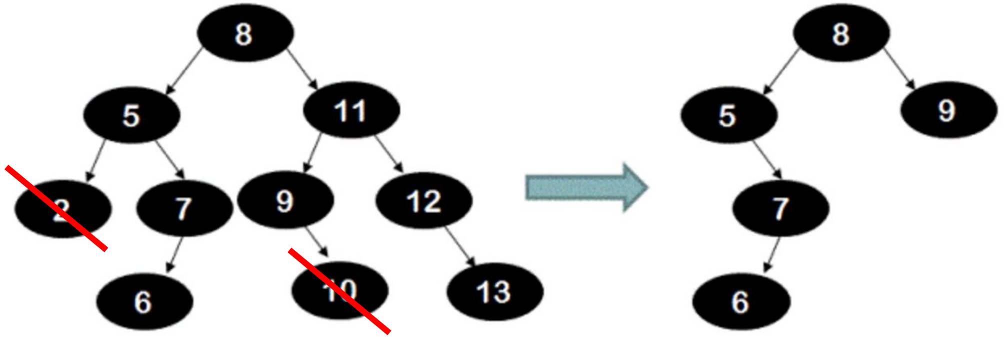
`keepRange(3,9)`



BST Exercise I

Write a BinaryTree member function `BinaryTree::keepRange(int min, int max)` that removes all nodes outside of that range (inclusive) and maintains Binary Tree structure.

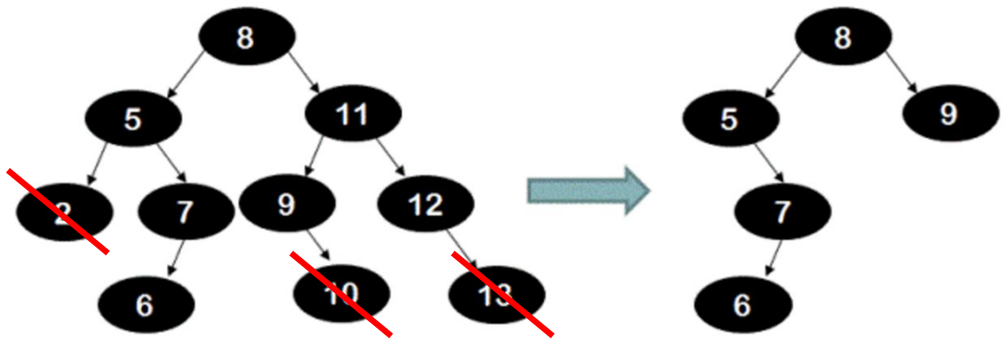
`keepRange(3,9)`



BST Exercise I

Write a BinaryTree member function `BinaryTree::keepRange(int min, int max)` that removes all nodes outside of that range (inclusive) and maintains Binary Tree structure.

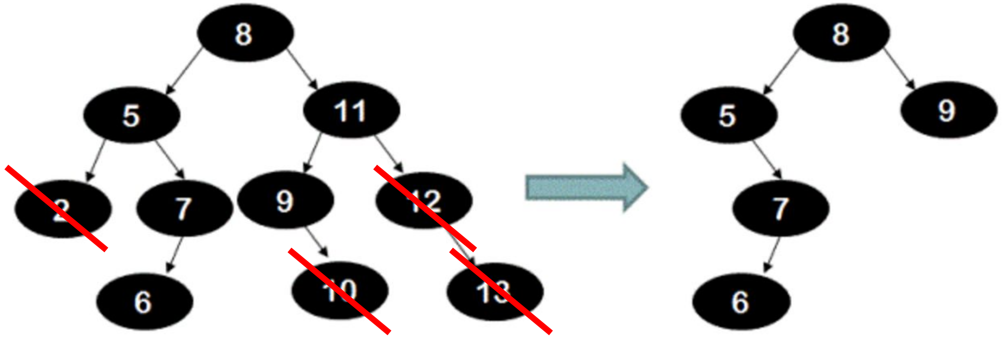
`keepRange(3,9)`



BST Exercise I

Write a BinaryTree member function `BinaryTree::keepRange(int min, int max)` that removes all nodes outside of that range (inclusive) and maintains Binary Tree structure.

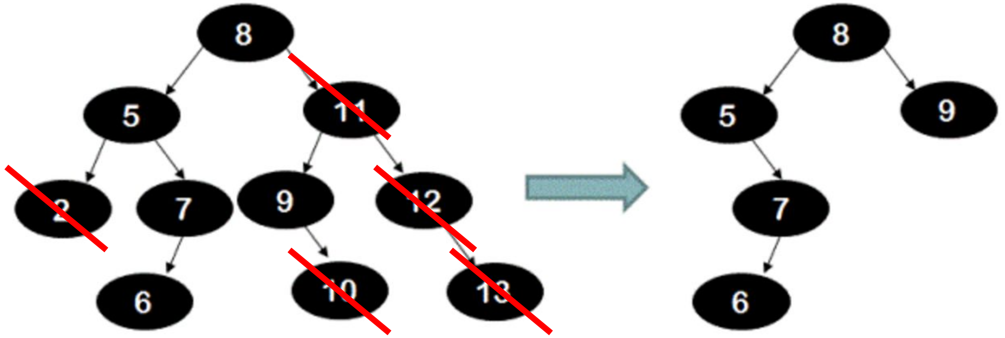
`keepRange(3,9)`



BST Exercise I

Write a BinaryTree member function `BinaryTree::keepRange(int min, int max)` that removes all nodes outside of that range (inclusive) and maintains Binary Tree structure.

`keepRange(3,9)`



BST Exercise I - **Remove from leaves up!**

```
void BinaryTree::keepRange(int min, int max) {
     keepRangeHelper(root, min, max);
}
void keepRangeHelper(TreeNode*& currentNode, int min, int max)) {
      if (currentNode != NULL) {
            keepRangeHelper(currentNode->left, min, max);
            keepRangeHelper(currentNode->right, min, max);
            if (currentNode->value < min) {
                  currentNode = currentNode->right;
            }
            if (currentNode->value > max) {
                  currentNode = currentNode->left;
            }
      }
}
```

Program logic:

- We need a helper function!
- We can remove leaves from the bottom up
- We can use *& to modify the actual pointer stored in the tree structure
- **Challenge:** Do this without recursion!
- **What about memory??**

BST Exercise I - **Remove from leaves up!**

```
void BinaryTree::keepRange(int min, int max) {
     keepRangeHelper(root, min, max);
}
void keepRangeHelper(TreeNode*& currentNode, int min, int max)) {
      if (currentNode != NULL) {
            keepRangeHelper(currentNode->left, min, max);
            keepRangeHelper(currentNode->right, min, max);
            if (currentNode->value < min) {
                  TreeNode* trash = currentNode;
                  currentNode = currentNode->right;
                  delete trash;
            }
            if (currentNode->value > max) {
                  TreeNode* trash = currentNode;
                  currentNode = currentNode->left;
                  delete trash;
            }
      }
}
```

Program logic:

- We need a helper function!
- We can remove leaves from the bottom up
- We can use *& to modify the actual pointer stored in the tree structure
- **Challenge:** Do this without recursion!

Heaps!

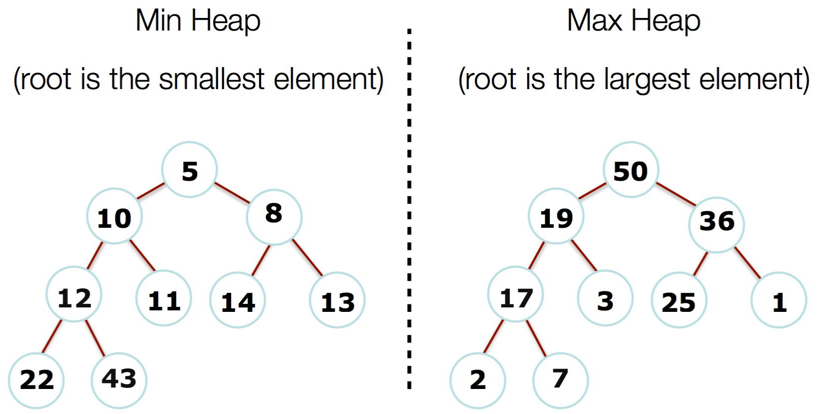
Heaps

A heap is a tree-based structure that satisfies the heap property: Parents have a higher priority key than any of their children.

Heaps are completely filled, with the exception of the bottom level. (always balanced)

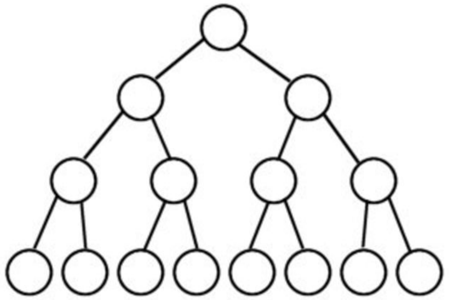
Often stored in arrays

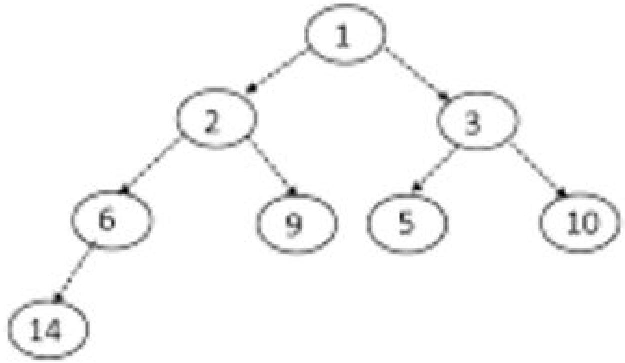
These are binary heaps -->



Heap Exercise I

What are the minimum and maximum numbers of elements in a heap of height *h*?

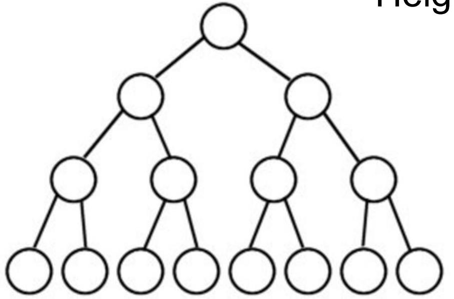




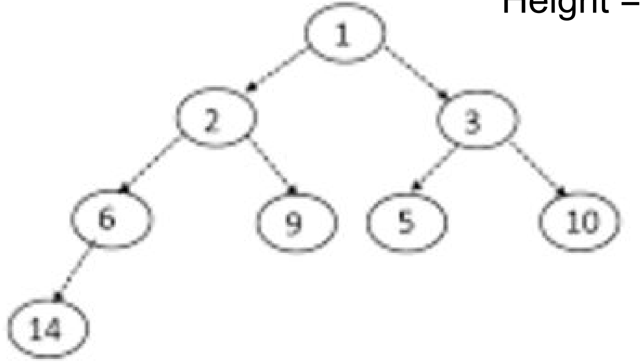
Heap Exercise I

What are the minimum and maximum numbers of elements in a heap of height *h*?

Height = 3



Height = 3



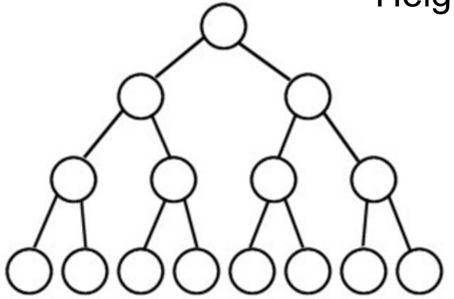
Heap Exercise I

What are the minimum and maximum numbers of elements in a heap of height *h*?

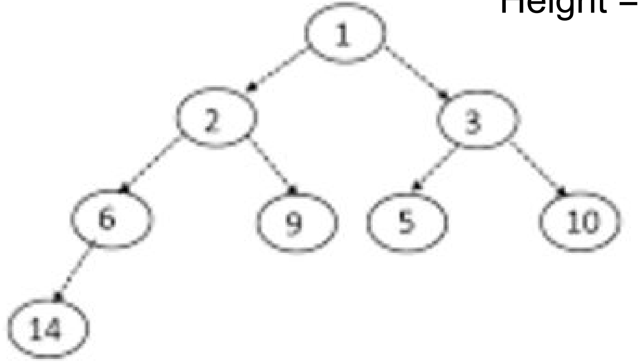
(max if complete) $1+2+2^2+2^3+\ldots+2^h=2^{h+1}-1$

(min if the lowest level has just 1 element and the other levels are complete) $2^h-1+1=2^h$

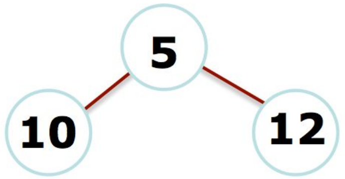
Height = 3

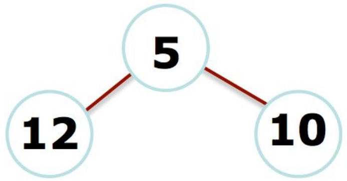


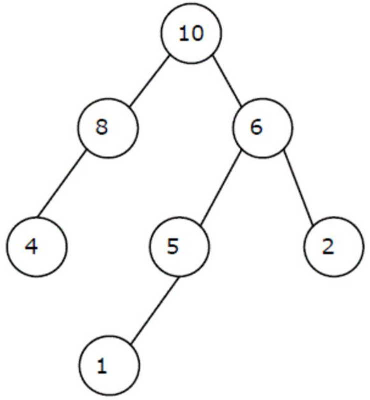
Height = 3

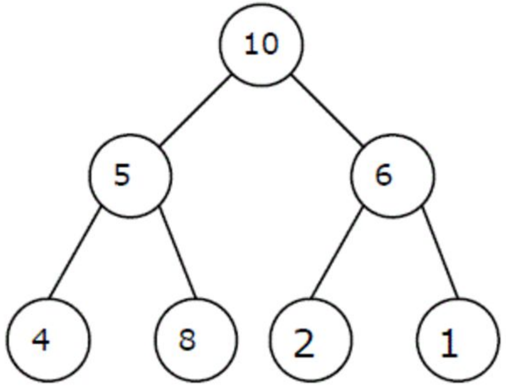


Heap Exercise II: Is it a Heap?



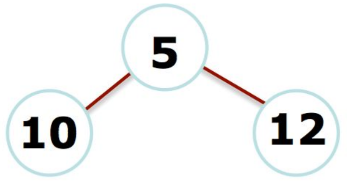


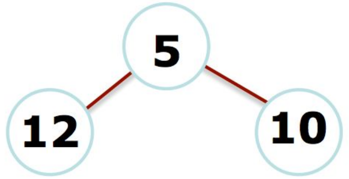


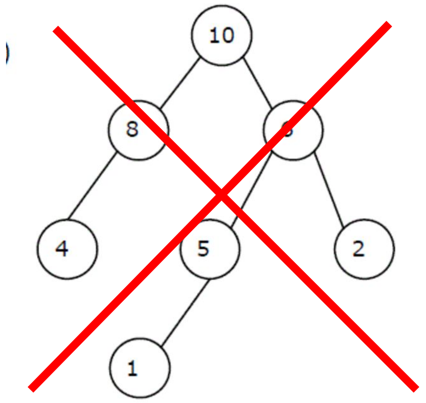


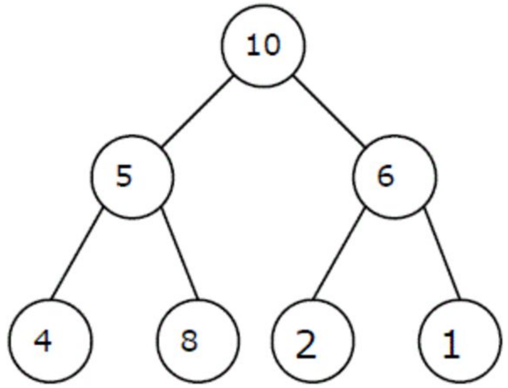
	5	10	8	12	11	14	13	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Heap Exercise II: Is it a Heap?









	5	10	8	12	11	14	13	22	43		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Graphs & Graph Algorithms

Graphs

Like a tree but with NO RULES!

Types of Graphs:

- Directed vs. Undirected
- Weighted vs. Unweighted
- Cyclic vs. Acyclic
- Connected
- Complete

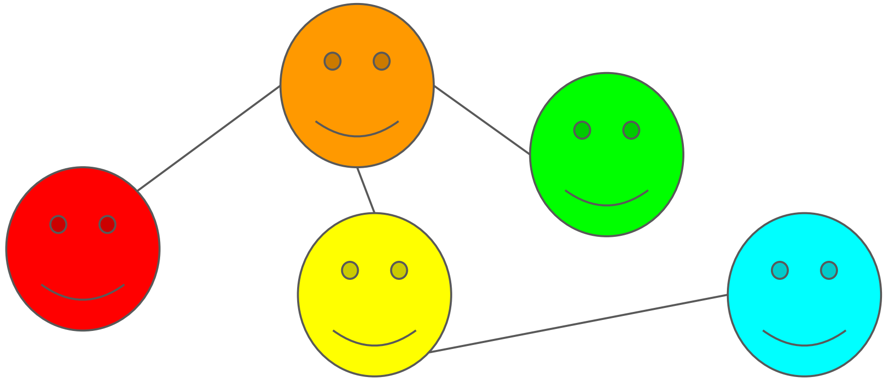
```
struct Node{
  string value;
  Vector<Edge *> edges;
};

struct Edge{
  Node * start;
  Node * end;
};

struct Graph{
  Set<Node *> nodes;
  Set<Edge*> edges;
};
```

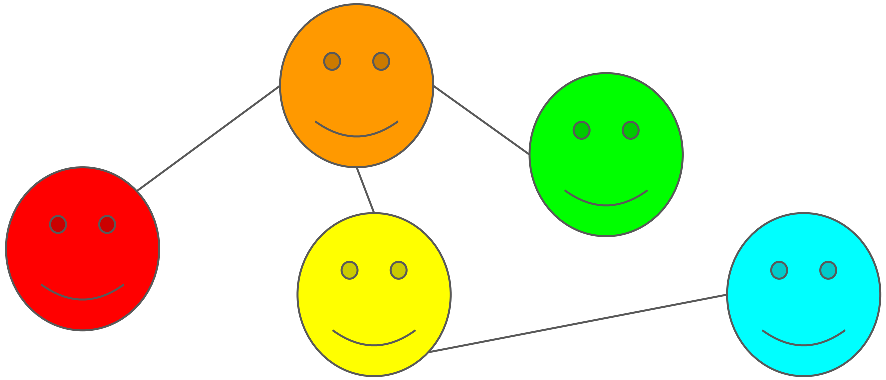
Graph Exercise I

Write a BasicGraph member function bool `BasicGraph::isFriendOfFriend(Vertex* friend1, Vertex* friend2)` that returns true if friend1 and friend2 share a mutual friend.



Graph Exercise I

Write a BasicGraph member function bool `BasicGraph::isFriendOfFriend(Vertex* friend1, Vertex* friend2)` that returns true if friend1 and friend2 share a mutual friend.



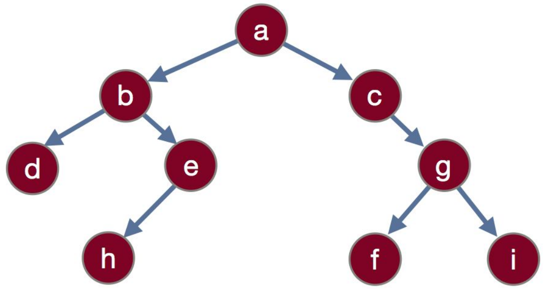
red and green? TRUE
red and blue? FALSE
red and orange? FALSE
red and red? FALSE

Graph Exercise I - **Remember edge cases!**

```
bool isFriendOfFriend(BasicGraph& graph, Vertex* friend1, Vertex* friend2) {
      if (friend1 == friend2) {
            return false;
      }
      for (Vertex* mutual_friend : graph.getNeighbors(friend1)) {
            for (Vertex* friendsfriend : graph.getNeighbors(mutual_friend)) {
                  if (friendsfriend == friend2) {
                        return true;
                  }
            }
      }
      return false;
}
```

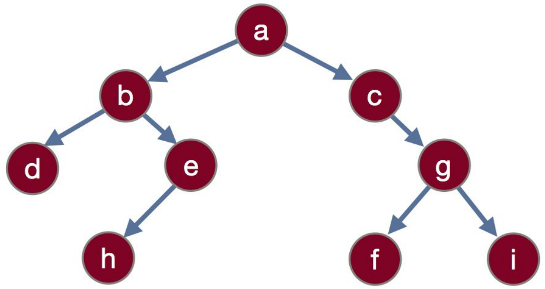
Challenge: Generalize this to take in a parameter n which represents how many degrees of separation must be between friend1 and friend2!

Graph Algorithms



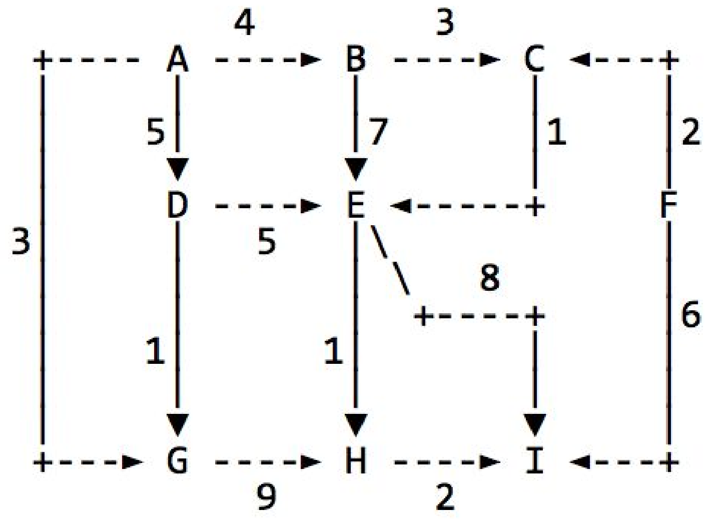
- BFS vs DFS
    - **Exercise**: Which is guaranteed to find the shortest path?
- What if paths have costs?
    - Dijkstra (like BFS but PQ instead of Queue)
    - A* (like Dijkstra but with a heuristic added to path priority

Graph Algorithms



- BFS vs DFS
    - **Exercise**: Which is guaranteed to find the shortest path? **BFS**
- What if paths have costs?
    - Dijkstra (like BFS but PQ instead of Queue)
    - A* (like Dijkstra but with a heuristic added to path priority

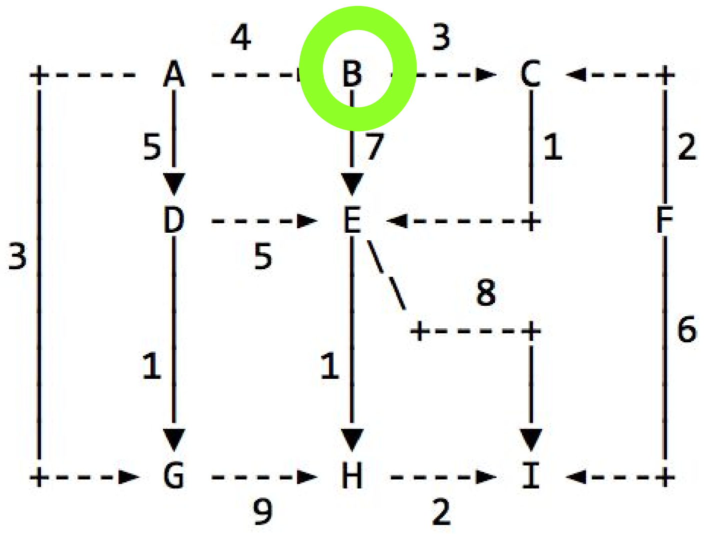
Graph Exercise II: Tracing



In what order would BFS mark the nodes as visited when searching for a path from B to I?

What path would be returned?

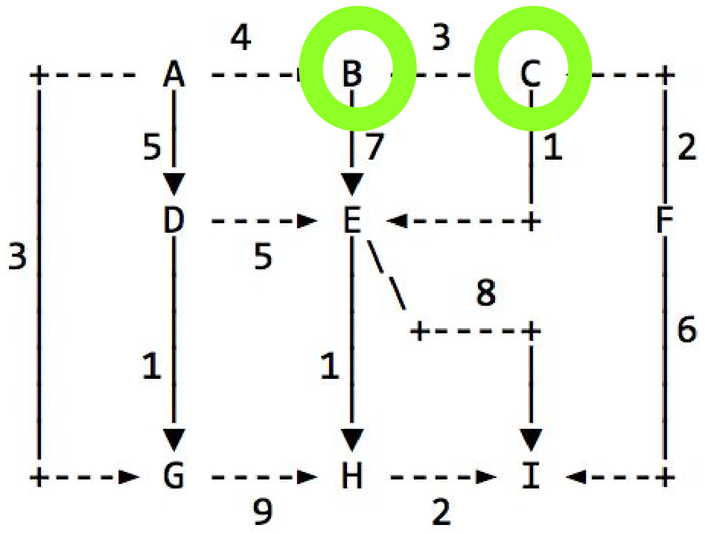
Graph Exercise II: Tracing



In what order would BFS mark the nodes as visited when searching for a path from B to I?

What path would be returned?

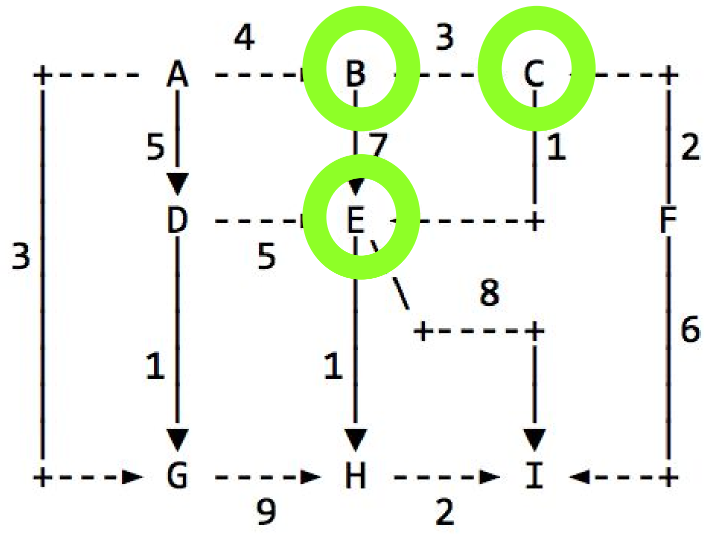
Graph Exercise II: Tracing



In what order would BFS mark the nodes as visited when searching for a path from B to I?

What path would be returned?

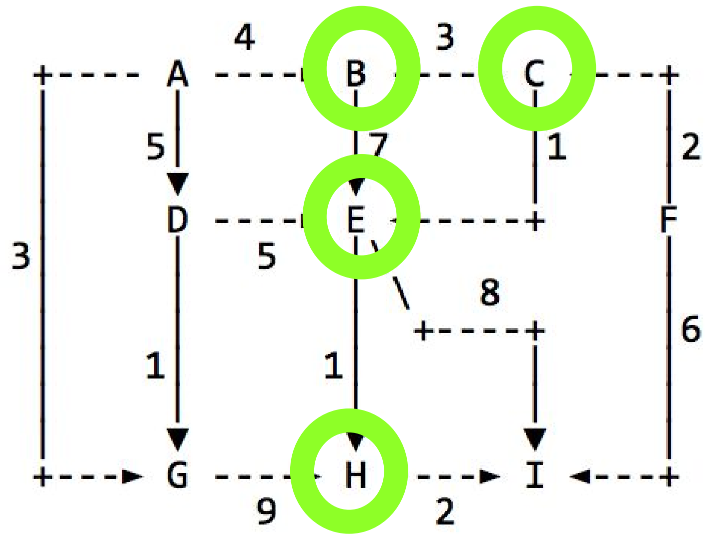
Graph Exercise II: Tracing



In what order would BFS mark the nodes as visited when searching for a path from B to I?

What path would be returned?

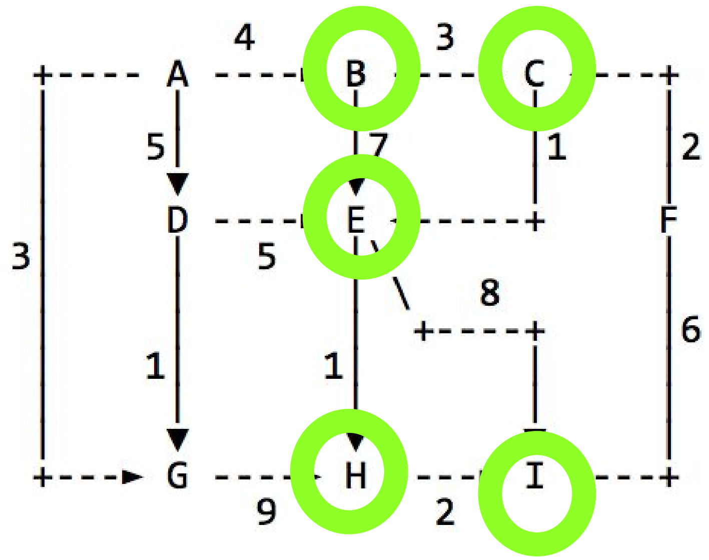
Graph Exercise II: Tracing



In what order would BFS mark the nodes as visited when searching for a path from B to I?

What path would be returned?

Graph Exercise II: Tracing



In what order would BFS mark the nodes as visited when searching for a path from B to I?
{B,C,E,H,I}

What path would be returned?
{B,E,I}

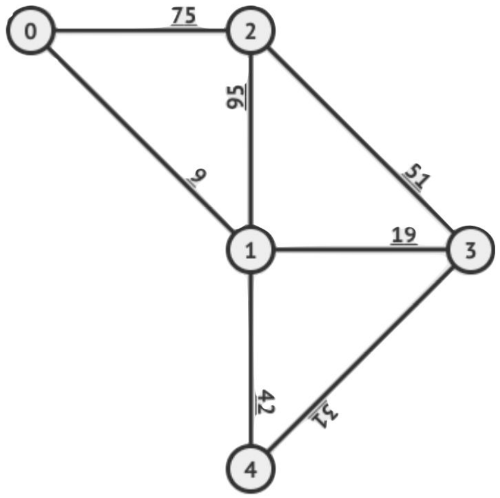
Types of Graph Problems

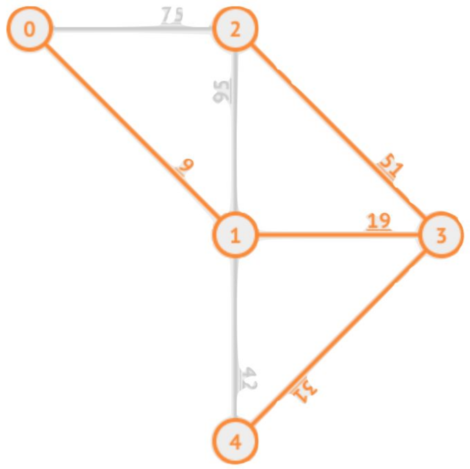
- Tracing algorithms (ex: in which order does DFS find the nodes)
- Testing a property (ex: isReachable)
- Building up a collection (ex: find all friends n degrees of separation away)
- And more!

Minimum Spanning Trees

Minimum Spanning Tree

Definition: A **Spanning Tree (ST)** of a connected undirected weighted graph **G** is a subgraph of **G** that is a **tree** and **connects (spans) all vertices of G**. A graph **G** can have multiple STs. A **Minimum Spanning Tree (MST)** of **G** is a ST of **G** that has the **smallest total weight** among the various STs. A graph **G** can have multiple MSTs but the MST weight is unique.





Minimum Spanning Tree

*Algorithm to find one: Kruskal’s

More practice

For more practice, check out section handouts 7 and 8 as well as the lectures and section handouts from cs106x, located at cs106x.stanford.edu