106B Final Review Session

Slides by Sierra Kaplan-Nelson and Kensen Shi
Livestream managed by Jeffrey Barratt

Topics to Cover

- Sorting
- Searching
- Heaps and Trees
- Graphs (with Recursive Backtracking)
- Inheritance / Polymorphism
- Linked Lists
- Big-O (sprinkled throughout)

Sorting

Selection sort: Builds a sorted list one element at a time by repeatedly selecting the minimum in the unsorted part and swapping it with left-most unsorted element.

Big-O?

```
void selectionSort(Vector& vec) {
    int n = vec.size(); // already-fully-sorted section grows
                        // one at a time from left to right
    for (int left = 0; left < n; left++) {
        int right = left; // find the min element in the entire unsorted section
        for (int i = left + 1; i < n; i++) {
            // found new min?
            if (vec[i] < vec[right]) right = i;
        }
        // swap min into sorted section
        int tmp = vec[left];
        vec[left] = vec[right];
        vec[right] = tmp;
    }
}
```

Sorting

Selection sort: Builds a sorted list one element at a time by repeatedly selecting the minimum in the unsorted part and swapping it with left-most unsorted element.

Big-O? **$O(N^2)$**

```
void selectionSort(Vector& vec) {
    int n = vec.size(); // already-fully-sorted section grows
                        // one at a time from left to right
    for (int left = 0; left < n; left++) {
        int right = left; // find the min element in the entire unsorted section
        for (int i = left + 1; i < n; i++) {
            // found new min?
            if (vec[i] < vec[right]) right = i;
        }
        // swap min into sorted section
        int tmp = vec[left];
        vec[left] = vec[right];
        vec[right] = tmp;
    }
}
```

Sorting

Insertion sort: Builds sorted list one element at a time by taking each element and swapping it with its left neighbor until it's in the right place.

Big-O?

```
void insertionSort(Vector& vec) {
    int n = vec.size();
    // already-sorted section grows one at a time from left to right
    for (int i = 1; i < n; i++) {
        int j = i; // does this item need to move left to be in order?
        while (j > 0 && vec[j-1] > vec[j]) {
            // keep swapping this item with its left neighbor if it is smaller
            int tmp = vec[j-1];
            vec[j-1] = vec[j];
            vec[j] = tmp;
            j--;
        }
    }
}
```

Sorting

Insertion sort: Builds sorted list one element at a time by taking each element and swapping it with its left neighbor until it's in the right place.

Big-O? **$O(N^2)$**

```
void insertionSort(Vector& vec) {
    int n = vec.size();
    // already-sorted section grows one at a time from left to right
    for (int i = 1; i < n; i++) {
        int j = i; // does this item need to move left to be in order?
        while (j > 0 && vec[j-1] > vec[j]) {
            // keep swapping this item with its left neighbor if it is smaller
            int tmp = vec[j-1];
            vec[j-1] = vec[j];
            vec[j] = tmp;
            j--;
        }
    }
}
```

Sorting

Bubble sort: Continuously goes through the vector and swaps each pair of items if they are in the wrong order until no more swaps are necessary.

Big-O?

```
void bubbleSort(Vector &vec) {
    bool swapped = true;                    // set flag to true to start first pass
    int n = vec.size();
    for (int i = 0; i < n && swapped; i++) {  // at most n passes needed, can stop early if no swaps
        swapped = false;
        for (int j = 0; j < n - 1; j++) {
            if (num[j+1] < vec[j]) {          // if swap needed
                int temp = vec[j];            // swap elements
                vec[j] = vec[j+1];
                vec[j+1] = temp;
                swapped = true;               // remember that a swap occurred
            }
        }
    }
}
```

Sorting

Bubble sort: Continuously goes through the vector and swaps each pair of items if they are in the wrong order until no more swaps are necessary.

Big-O? **$O(N^2)$**

```
void bubbleSort(Vector &vec) {
    bool swapped = true;                    // set flag to true to start first pass
    int n = vec.size();
    for (int i = 0; i < n && swapped; i++) {  // at most n passes needed, can stop early if no swaps
        swapped = false;
        for (int j = 0; j < n - 1; j++) {
            if (num[j+1] < vec[j]) {          // if swap needed
                int temp = vec[j];            // swap elements
                vec[j] = vec[j+1];
                vec[j+1] = temp;
                swapped = true;               // remember that a swap occurred
            }
        }
    }
}
```

Famously bad: Obama makes bubble sort joke

Sorting

Summary so far: All have the same (bad) big-O but in real life have certain trade offs and can be used in certain situations (especially insertion sort).

Sorting

Heap sort: “Improved selection sort” -- no longer takes linear time to find the min because you use insert items into a min heap.

Big-O?

Pseudo-code:

1. Take the unsorted array and insert each element into a heap priority queue
2. While the queue is not empty, dequeue an element from the heap priority queue

Sorting

Heap sort: “Improved selection sort” -- no longer takes linear time to find the min because you use insert items into a min heap.

Big-O? **O(N log N)**

Pseudo-code:

1. Take the unsorted array and insert each element into a heap priority queue
2. While the queue is not empty, dequeue an element from the heap priority queue

Sorting

Merge sort: Divide and conquer. Split the list into two halves, recurse on each half, and then merge the two sorted halves.

Big-O?

Pseudo-code:

1. Split the list into two halves.
2. Sort each half by recursing.
3. Merge the two sorted halves.

Sorting

Merge sort: Divide and conquer. Split the list into two halves, recurse on each half, and then merge the two sorted halves.

Big-O? **O(N log N).**

Intuition: Merging two sorted lists takes O(N) time, and the recursion goes O(log N) levels deep.

Pseudo-code:

1. Split the list into two halves.
2. Sort each half by recursing.
3. Merge the two sorted halves.

Sorting

Quick sort: Divide and conquer. Partition the list into two parts, and sort each part.

Big-O?

Pseudo-code:

1. Pick an element, called a pivot, from the array.
2. Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position.
3. Recurse on the left partition and the right partition.

Sorting

Quick sort: Divide and conquer. Partition the list into two parts, and sort each part.

Big-O? **O(N log N),** although in some extremely bad cases... $O(N^2)$ if pivot is smallest or largest element in list.

Pseudo-code:

1. Pick an element, called a pivot, from the array.
2. Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position.
3. Recurse on the left partition and the right partition.

Sorting Summary (https://www.youtube.com/watch?v=kPRA0W1kECg - first 2 minutes)

	Best	Average	Worst	Intuition
Selection	$O(N^2)$	$O(N^2)$	$O(N^2)$	Select min among unsorted elements, swap to left
Insertion	O(N)	$O(N^2)$	$O(N^2)$	Insert elements into sorted portion by pushing them leftward
Bubble	O(N)	$O(N^2)$	$O(N^2)$	Swap adjacent out-of-order elements until no swaps needed
Heap	O(N log N)	O(N log N)	O(N log N)	Put everything into a heap and dequeue one-by-one
Merge	O(N log N)	O(N log N)	O(N log N)	Split into halves, recurse on each half, merge sorted halves
Quick	O(N log N)	O(N log N)	$O(N^2)$	Pick pivot, partition, recurse on partitions

Sorting Summary

If they are already sorted (or almost sorted)

If the pivot happens to be the greatest or least element each time

	Best	Average	Worst	Algorithm
Selection	$O(N^2)$	$O(N^2)$	$O(N^2)$	Select min among unsorted elements, swap to left
Insertion	O(N)	$O(N^2)$	$O(N^2)$	Insert elements into sorted portion by pushing them leftward
Bubble	O(N)	$O(N^2)$	$O(N^2)$	Swap adjacent out-of-order elements until no swaps needed
Heap	O(N log N)	O(N log N)	O(N log N)	Put everything into a heap and dequeue one-by-one
Merge	O(N log N)	O(N log N)	O(N log N)	Split into halves, recurse on each half, merge sorted halves
Quick	O(N log N)	O(N log N)	$O(N^2)$	Pick pivot, partition, recurse on partitions

Searching

Linear search: Look through all elements in order, until you find the one you want.

Big-O: O(N)

Requirements: None

Binary search: Look at the middle. If that matches what you want, then stop. Otherwise, if you want something greater, look to the right, and if you want something smaller, look to the left.

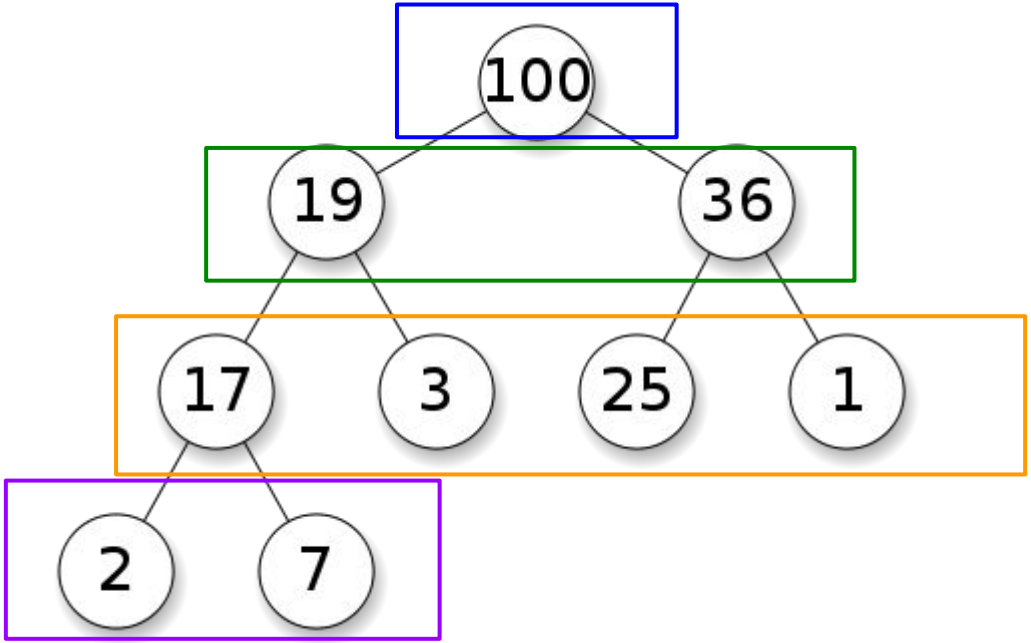
Big-O: O(log N)

Requirements: List must be previously sorted

Heaps & Trees

- Min/Max Heap
- Binary Trees vs. Binary Search Trees
- In-Order, Pre-Order, and Post-Order Traversals

Max Heap Example



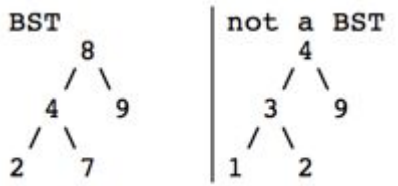
1	2	3	4	5	6	7	8	9
100	19	36	17	3	25	1	2	7

Max Heap Example (on board)

What would a max heap look like if inserted (in order) 10, 16, 3, 12, 20, 6, 5, 13 ?

Trees Problem: is BST (on board)

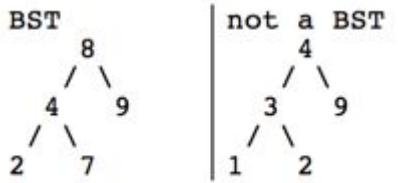
isBST. Write a member function isBST that returns whether or not a binary tree is arranged in valid binary search tree (BST) order. Remember that a BST is a tree in which every node *n*'s left subtree is a BST that contains only values less than *n*'s data, and its right subtree is a BST that contains only values greater than *n*'s data.



Trees Problem: is BST (on board)

Intuition: think of a tree as a list -- what are the properties of the list if it is a valid BST?

2 4 7 8 9 1 3 2 4 9



Trees Problem: is BST (on board)

```
bool BinaryTree::isBST() {
    TreeNode* prev = NULL;
    return isBST(root, prev);
}

// An in-order walk of the tree, storing the last visited node in 'prev'
bool BinaryTree::isBST(const TreeNode*& node, TreeNode*& prev) {
    if (node == NULL) {
        return true;
    } else if (!isBST(node->left, prev) || (prev != NULL && node->data <= prev->data)) {
        return false;
    } else {
        prev = node;
        return isBST(node->right, prev);
    }
}
```

Big-O?

Trees Problem: is BST (on board)

```
bool BinaryTree::isBST() {
    TreeNode* prev = NULL;
    return isBST(root, prev);
}

// An in-order walk of the tree, storing the last visited node in 'prev'
bool BinaryTree::isBST(const TreeNode*& node, TreeNode*& prev) {
    if (node == NULL) {
        return true;
    } else if (!isBST(node->left, prev) || (prev != NULL && node->data <= prev->data)) {
        return false;
    } else {
        prev = node;
        return isBST(node->right, prev);
    }
}
```

Big-O? O(N)

(all traversals are O(N) because we look at every element exactly once)

Graph Problem: longestPath (on board)

longestPath. Write a function longestPath that returns the cost of the longest path in a directed graph. The graph will have nonnegative edge costs. A path may not visit a vertex more than once, and does not have to visit all vertices. **Use recursive backtracking**.

```
double longestPath(BasicGraph& graph) {

}
```

Graph Problem: longestPath (on board)

```
double longestPathHelper(BasicGraph& graph, Vertex* cur, Set<Vertex*> visited) {
    double maxLength = 0.0;
    for (Vertex* neighbor : graph.getNeighbors(cur)) {
        if (visited.contains(neighbor)) continue;

        visited.add(neighbor);
        double length = graph.getEdge(cur, neighbor)->cost + longestPathHelper(graph, neighbor, visited);
        maxLength = max(maxLength, length);
        visited.remove(neighbor);
    }
    return maxLength;
}

double longestPath(BasicGraph& graph) {
    double maxLength = 0.0;
    for (Vertex* v : graph.getVertexSet()) {
        Set<Vertex*> visited;
        visited.add(v);
        maxLength = max(maxLength, longestPathHelper(graph, v, visited));
    }
    return maxLength;
}
```

Graph Problem: longestPath, version 2 (on board)

longestPath. What if we now want to compute the longest path between a given start and end vertex? What changes need to be made?

```
double longestPath(BasicGraph& graph, Vertex* start, Vertex* end) {

}
```

Graph Problem: longestPath, version 2 (on board)

```
double longestPathHelper(BasicGraph& graph, Vertex* cur, Vertex* end, Set<Vertex*>& visited) {
   if (cur == end) return 0.0;

   double maxLength = 0.0;
   for (Vertex* neighbor : graph.getNeighbors(cur)) {
       if (visited.contains(neighbor)) continue;

       visited.add(neighbor);
       double length = graph.getEdge(cur, neighbor)->cost + longestPathHelper(graph, neighbor, end, visited);
       maxLength = max(maxLength, length);
       visited.remove(neighbor);
   }
   return maxLength;
}

double longestPath(BasicGraph& graph, Vertex* start, Vertex* end) {
   Set<Vertex*> visited;
   visited.add(start);
   return longestPathHelper(graph, start, end, visited);
}
```

Inheritance & Polymorphism

- Compile-time vs runtime
- A caste on a class may be necessary for the compiler but will not change the runtime call of a virtual method.
- A caste will change the behavior of calling a non-virtual method.
- How can you get a runtime crash?

Inheritance & Polymorphism

Steps for Inheritance / Polymorphism Problems:

```
BaseType* var = new DerivedType();
```

0a. Draw the inheritance tree.
0b. Propagate “virtualness” downward.
1. **Start at the base class** (if called from an object) **or the calling class** (if called from a class), and go upward until you find the method.
2a. If that method is **not virtual**, use it.
2b. If that method is **virtual**, then go to the **derived class**, and go upward until you find the method, and use that one.

Other notes:

1. If you ever run off the inheritance tree without finding the method, it’s a compiler error.
2. If the derived type doesn’t inherit from the base type, it’s a compiler error.
3. Casting effectively changes the base type.
4. If you cast to an invalid new base type, the compiler won’t complain, but the behavior is undefined (probably a crash).

Inheritance & Polymorphism

```
class A {
public:
   virtual void m1() {
       cout << "a1 ";
   }
   void m3() {
       cout << "a3 ";
       m1();
   }
   void m4() {
       cout << "a4 ";
   }
};
```

```
class B : public A {
public:
   virtual void m1() {
       cout << "b1 ";
       m4();
   }
   virtual void m2() {
       cout << "b2 ";
       m4();
   }
   void m5() {
       cout << "b5 ";
   }
};
```

```
class C : public B {
public:
   void m1() {
       cout << "c1 ";
       B::m2();
   }
   void m2() {
       cout << "c2 ";
       m4();
   }
   void m4() {
       cout << "c4 ";
   }
};
```

Execute the following code:

```
A* v1 = new B();
v1->m1(); cout << endl;
v1->m3(); cout << endl;
v1->m5(); cout << endl;
((B*)v1)->m5(); cout << endl;

A* v2 = new C();
v2->m1(); cout << endl;
v2->m4(); cout << endl;
((C*)v2)->m4(); cout << endl;

B* v3 = new B();
((A*)v3)->m4(); cout << endl;

C* v4 = new B();
v4->m2();

A* v5 = new B();
((C*)v5)->m1();
```

Inheritance & Polymorphism

```
class A {
public:
   virtual void m1() {
       cout << "a1 ";
   }
   void m3() {
       cout << "a3 ";
       m1();
   }
   void m4() {
       cout << "a4 ";
   }
};
```

```
class B : public A {
public:
   virtual void m1() {
       cout << "b1 ";
       m4();
   }
   virtual void m2() {
       cout << "b2 ";
       m4();
   }
   void m5() {
       cout << "b5 ";
   }
};
```

```
class C : public B {
public:
   void m1() {
       cout << "c1 ";
       B::m2();
   }
   void m2() {
       cout << "c2 ";
       m4();
   }
   void m4() {
       cout << "c4 ";
   }
};
```

Execute the following code:

```
A* v1 = new B();
v1->m1(); cout << endl;          b1 a4
v1->m3(); cout << endl;          a3 b1 a4
v1->m5(); cout << endl;          Compiler Error
((B*)v1)->m5(); cout << endl;    b5

A* v2 = new C();
v2->m1(); cout << endl;          c1 b2 a4
v2->m4(); cout << endl;          a4
((C*)v2)->m4(); cout << endl;    c4

B* v3 = new B();
((A*)v3)->m4(); cout << endl;    a4

C* v4 = new B();                 Compiler Error
v4->m2();                        (error from above)

A* v5 = new B();
((C*)v5)->m1();                  Undefined behavior (crash)
```

Determine output

```
class Animal {
	public:
	virtual void food() = 0;

	virtual void habitat() {
		cout << "A habitat" << endl;
	}
};

class Mammal : public Animal {
	public:
	virtual void food() {
		cout << "M food" << endl;
	}
	virtual void habitat() {
		Animal::habitat();
		cout << "M habitat" << endl;
	}
	virtual void behavior() {
		cout << "M behavior" << endl;
	}
};
```

```
class Reptile : public Animal {
	public:
	virtual void food() {
		cout << "R food" << endl;
	}
	virtual void size() {
		cout << "R size" << endl;
	}
};

class Dog : public Mammal {
	public:
	virtual void habitat() {
		cout << "D habitat" << endl;
	m1();
	}
	virtual void size() {
		cout << "D size" << endl;
	}
};
```

```
Animal* var1 = new Reptile();
Mammal* var2 = new Mammal();
Mammal* var3 = new Dog();
Reptile* var4 = new Reptile();
Animal* var1 = new Animal();

var1->size();

var1->habitat();

var2->habitat();

var3->food();

var3->size();

var4->habitat();

((Dog*)var3)->size();

((Reptile*)var2)->size();
```

Determine output

```
class Animal {
	public:
	virtual void food() = 0;

	virtual void habitat() {
		cout << "A habitat" << endl;
	}
};

class Mammal : public Animal {
	public:
	virtual void food() {
		cout << "M food" << endl;
	}
	virtual void habitat() {
		Animal::habitat();
		cout << "M habitat" << endl;
	}
	virtual void behavior() {
		cout << "M behavior" << endl;
	}
};
```

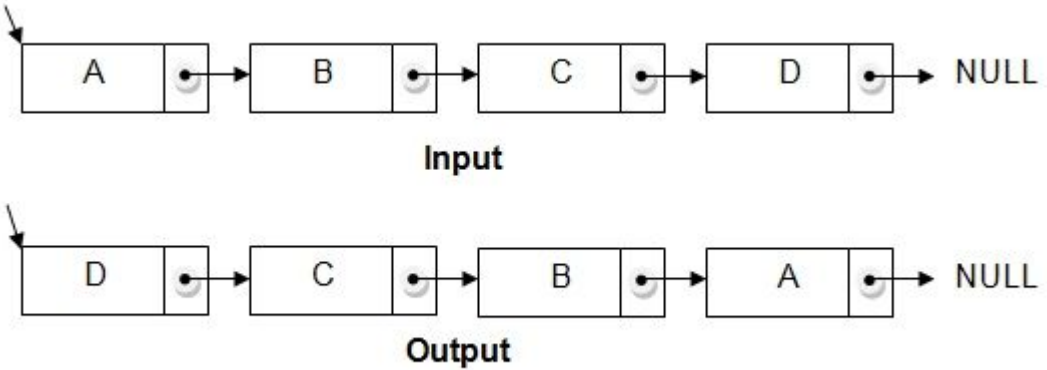
```
class Reptile : public Animal {
	public:
	virtual void food() {
		cout << "R food" << endl;
	}
	virtual void size() {
		cout << "R size" << endl;
	}
};

class Dog : public Mammal {
	public:
	virtual void habitat() {
		cout << "D habitat" << endl;
		m1();
	}
	virtual void size() {
		cout << "D size" << endl;
	}
};
```

```
Animal* var1 = new Reptile();
Mammal* var2 = new Mammal();
Mammal* var3 = new Dog();
Reptile* var4 = new Reptile();
Animal* var1 = new Animal();


var1->size(); //Compiler error

var1->habitat(); A habitat

var2->habitat(); A habitat M habitat

var3->food(); //Compiler error

var3->size(); //Compiler error

var4->habitat(); A habitat

((Dog*)var3)->size(); D size //Cast makes
compile

((Reptile*)var2)->size(); RUNTIME ERROR
```

Linked Lists

Problem: Reverse a singly linked list (on board)

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



Linked Lists

```
void LinkedList::reverseLinkedList() {
    if(head == NULL) return;

    Node *prev = NULL;
    Node *current = head;
    Node *next = NULL;
    while(current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    // now let the head point at the last node (prev)
    head = prev;
}
```

Other Questions?