Aut19 CS106B Final Review

Based on content by previous CS106B instructors & Head TAs
Exam Logistics

Date: Monday, Dec 9th at 8:30am

Location: Cubberley & Dinkelspiel (separated by last name - please check website)

Open book, 1 double-side sheet of notes

Please write darkly & clearly
Exam Topics

**Everything** covered in section, in lecture and the assignments is fair game

There will be an emphasis on the second half of the course, but the exam will also include material tested on the midterm
Exam Topics (since Midterm)

- **Classes & Objects**: public vs private, constructor, deconstructor, operator overloading
- **Priority Queues & Heaps**: (un)sorted array implementation, binary tree, binary heap (complete & heap property), binary heap in an array, enqueue & bubble up, dequeue & trickle down
- **Sorting**: selection sort, bubble sort, insertion sort, heap sort, mergesort, quicksort
- **Memory & Pointers**: statically vs dynamically-allocated arrays, new/delete, struct, pointer
- **Linked Lists**: next pointer, prev pointer, insert node, remove node
- **Trees**: definition of BST, insert in BST, Tree structures that maintain balance, pre/post/in order tree traversals, Huffman Trees
- **Graphs**: node, edge, un(directed) graph, path, neighbor/adjacent, connected, reachable, cycle, adjacency matrix, BFS, Dijkstra’s shortest path
- **Hashing**: hash function, hash table inserts, hash collisions
- **Lexicon Overview**
Exam Tips

1. Take a least 1 practice exam in exam conditions

2. When studying, write your answers and before looking at the solutions manually trace through the code on a few examples

3. Draw pictures! (especially for linked lists & memory diagram questions)

4. Know Big-Oh for algorithms covered in class (sorting functions, inserting into binary heap, etc)

5. Don’t write code until you can explain, in words, how you’re going to solve a problem

6. Be able to compare/contrast algorithms and implementations (i.e. when hash collisions occur)
Common Exam Questions

1. Drawing what a heap/BST would look like after inserting certain elements & then removing certain elements (make sure to be able to draw the array representing the heap as well)

2. Drawing a memory diagram given some code

3. Writing code that will transform a “before” picture of a linked list into an “after” picture

4. Tree Traversals
Classes & Objects
Classes & Objects: Terminology

- **Member Variables**: keep track of the state inside each object; each object has a copy of each member variable
- **Member functions**: define the behavior inside each object; also called methods; methods can interact with data inside the object
- **Constructor**: initializes new objects as they are created; sets the initial state of each new object
- **Deconstructor**: called when object is deleted by program (i.e. when object falls out of scope); useful if object needs to free memory as it dies (delete arrays & pointers)
- **.h file**: header file containing the interface (declarations)
- **.cpp file**: source file containing definitions (method bodies); content of the .h file is #included inside .cpp files
- **const**: a const reference parameter can’t be modified by the function; a const member function can’t change the object’s state
- **#ifndef / #define / #endif**: protection in case multiple .cpp files include the same .h, so that its contents aren’t declared twice; #ifndef and #define go at the top of the .h file and are followed by _classname_h; #endif goes at the bottom of the .h file
1. Member function declaration in .cpp
   returnType ClassName::methodName(parameters) {
   ... }

2. Know how to overload operators like `<<`
   Note that operators go outside the class’ closing } brace in .h file
   Pass in an object by reference as a parameter

3. In the deconstructor, always consider if you need to free memory!
Memory & Pointers
What does a pointer store? A memory address.

How do I declare a pointer?
```
int* ptr = new int(5);
int x = 6;
int* ptr2 = &x;
```

How do I get the value stored at the memory address the pointer stores?
Dereference the pointer using the *
```
*ptr = 7
```
Note for structs:
```
structPtr->memberVariable
*(structPtr).memberVariable
```
What is a dynamically-allocated array? It is an array on the heap. The variable on the stack is a pointer to the first element of the array. The allocated memory must be released, else you leak memory.

What is a statically-allocated array? It is an array on the stack. It can never be resized, but the memory does not need to be freed.

How do I get the value stored in the array?
The brackets [] automatically dereferences a pointer
```
int* arr = new int[5]();
arr[0] = 1
```
int season = 6;
bool result = false;
Baker* finalists = new Baker[2];
finalists -> recipes = season;
finalists[0].starBaker = nullptr;
finalists[0].won = &result;
finalists[1].starBaker = finalists;
finalists[1].won = &result;
result = true;

struct Baker {
    int recipes;
    Baker* starBaker;
    bool* won;
};
int season = 6;
bool result = false;

Stack

<table>
<thead>
<tr>
<th>season</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>false</td>
</tr>
</tbody>
</table>

Heap
Baker* finalists = new Baker[2];

Stack
- season: 6
- result: false
- finalists

Heap
- recipes
- starBaker
- won

- recipes
- starBaker
- won

0
1
finalists -> recipes = season;

Stack
- season: 6
- result: false
- finalists

Heap
- recipes: 6
- starBaker
- won
- recipes
- starBaker
- won

0
1
```c
finalists[0].starBaker = nullptr;
```
finalists[0].won = &result;

Stack

- season: 6
- result: false
- finalists

Heap

- recipes: 6
- starBaker
- won
- recipes
- starBaker
- won
- NULL
- 0
- 1
finalists[1].starBaker = finalists;

Stack
season 6
result false
finalists

Heap
recipes 6
starBaker NULL
won 0
recipes
starBaker
won 1
finalists[1].won = &result;
result = true;

Stack
- season: 6
- result: true
- finalists

Heap
- recipes: 6
- starBaker
  - won
- recipes
- starBaker
  - won
  - won
  - NULL
  - 0
  - 1
1. Remember that new and delete are always a pair when you’re writing your own code! Sometimes the new and delete are separated across different methods, but it is always there.

2. Always be cautious of uninitialized memory!
   ```cpp
type* name = new type[length] // uninitialized array
cout << name[0] // will print out garbage (random values)
```

3. Be really deliberate with memory diagrams and please do some from the practice exam.
Linked Lists
Linked List Overview

```c
ListNode* node = new ListNode(3);
node->next = new ListNode(5);
```
Linked List Overview

Read or examine a list:

```cpp
returnType name(ListNode* front, ...) {
    // example:
    void printList(ListNode* front) {
        ListNode* current = front;
        while (current != nullptr) {
            cout << current->data << endl;
            current = current->next;
        }
    }
}
```

Write or modify a list (notice the &):

```cpp
returnType name(ListNode*& front, ...) {
    // example:
    void removeFront(ListNode*& front) {
        if (front != nullptr) {
            ListNode* trash = front;
            front = front->next;
            delete trash;
        }
    }
}
```
Linked List Tips

1. Draw pictures! Make sure you know exactly what each line of code is doing and why.

2. Make sure to delete nodes that you no longer need but...

3. Make sure you don’t dereference nodes that you’ve deleted! (Hint: in pictures, helpful to cross out once deleted)

4. Always make sure that you have a pointer to every node that you want to access (having temporary pointers is a great strategy)

5. Common test cases to consider: empty list, list of length 1, list of length 2
Write code that will transform the before picture into the after **without creating any new nodes**. As a good way to check, swap your solution with a study buddy and trace through their solution while they trace through yours. When you trace through their solution, do you get the after picture? How about when they trace through yours?

Before

a) list

```
1 → 2 → 3
```

```
4 → 5
```

After

```
1 → 3 → 5
```

```
2 → 4
```
Trees
Each node has at most 2 children
Binary Search Tree

- Each node has at most 2 children
- All nodes to the left of the current node have values less than the current node’s value
- All nodes to the right of the current node have values greater than the current node’s value
1. Know the traversals
   a. Preorder traversal: do work on the current node, recurse left, recurse right
   b. Inorder traversal: recurse left, do work on the current node, recurse right
   c. Postorder traversal: recurse left, recurse right, do work on the current node

2. Before solving a tree problem, figure out which traversal works well with the problem

3. Traversals are recursive in nature
Write a function called partitionTree that takes a pointer to the root of a binary search tree as a parameter. That BST only contains positive values except for one mystery node, whose key is replaced by the value -1. partitionTree should return two lists: one of values less than the original value of the mystery node and one of values greater than the original value of the mystery node.

```c++
void partitionTree(BSTNode* root, Vector<int>& lessThan, Vector<int>& greaterThan){ ... }
```
Tree Problem: Example 1

lessThan: \{3, 6, 10, 15\}
greaterThan: \{27\}
Tree Problem: Example 2

lessThan: \{3, 6\}
greaterThan: \{10, 15, 23, 27\}
Tree Problem: Traversal

Which makes more sense: preorder, inorder or postorder search?

Put another way, in what order do we want to visit the nodes?
Tree Problem: Traversal

Which makes more sense: preorder, inorder or postorder search?

Put another way, in what order do we want to visit the nodes?

preorder: 15, 6, 3, 10, 23, 27

inorder: 3, 6, 10, 15, 23, 27

postorder: 3, 10, 6, 27, 23, 15
Tree Problem: Traversal

Which makes more sense: preorder, inorder or postorder search?

Put another way, in what order do we want to visit the nodes?

preorder: 15, 6, 3, 10, 23, 27
inorder: 3, 6, 10, 15, 23, 27
postorder: 3, 10, 6, 27, 23, 15
Tree Problem: Traversal

preorder: 15, 6, 3, -1, 23, 27
inorder: 3, 6, -1, 15, 23, 27
postorder: 3, -1, 6, 27, 23, 15

With an inorder traversal, all of the values before the mystery node have to be less than the original value of the mystery node and all of the values after the mystery node have to be greater than the original value of the mystery node.
Tree Problem: Pseudocode

Base case: current node is nullptr

Recurse on the left pointer

Do something with the current node

Recurse on the right pointer

To think about writing the code for this problem, first write the template for inorder tree traversal
Tree Problem: Pseudocode

Base case: current node is nullptr

Recurse on the left pointer

If the current value is -1: remember that we’ve found the mystery node

If the mystery node hasn’t been found: add current value to lessThan
Else: add current value to greaterThan

Recurse on the right pointer

What do we need to do at the current node? If the mystery node hasn’t been found yet, we add the value to lessThan. If it has been found we add the value to greaterThan
void partitionTree(BSTNode* root, Vector<int>& lessThan, Vector<int>& greaterThan){
    bool foundMystery = false;
    partitionTreeHelper(root, lessThan, greaterThan, foundMystery);
}

void partitionTreeHelper(BSTNode* root, Vector<int>& lessThan, Vector<int>& greaterThan, bool& foundMystery){
    if(root != nullptr) {
        partitionTreeHelper(root->left, lessThan, greaterThan, foundMystery);
        if(root->value == -1) {
            foundMystery = true;
        } else if (foundMystery) {
            greaterThan.add(root->value)
        } else {
            lessThan.add(root->value)
        }
    }
    partitionTreeHelper(root->right, lessThan, greaterThan, foundMystery);
}
Graphs
Graph Terminology

- Vertices and Edges
- Path: a sequence of edges that connect two nodes
- Cyclic vs. acyclic (is there a path from a vertex back to itself?)
- Directed vs. undirected (do the edges have a direction?)
- Connected: there is a path from each node to every other node
- Complete: there is an edge between every pair of nodes
- Weighted vs. unweighted (do the edges have a cost?)
Graph Algorithms

1. Be able to trace through breadth and depth first search on graphs
   a. Breadth first search finds the shortest path in terms of number of edges
      (uses a queue to implement)
   b. Depth first search is good at determining if a path exists between two nodes
      (uses recursion or a while loop to implement)

2. Know the benefits of Dijkstra’s algorithm over BFS
Appendix: Heaps
Heaps: Overview

A heap is a type of **complete binary tree** that obeys some ordering property.
(complete = each level is filled left to right before moving to the next level)

**Min-heap**: each parent is smaller than its children

**Max-heap**: each parent is larger than its children
Heaps: Stored as an array

Usually stored in an array instead of as a tree
Written level by level (so that for a node at index n, its parent is at n/2 and its children are at 2n and 2n+1 if we one-index the array)
Heaps vs BST

BST and Heap Facts (cheat sheet)

**Heap (Priority Queue)**
- **Structure:** must be “complete”
- **Order:** parent priority must be $\leq$ both children
  - This is for min-heap, opposite is true for max-heap
  - **No rule** about whether left child is $>$ or $<$ the right child
- **Big-O:** guaranteed $\log(n)$ enqueue and dequeue
- **Operations:** always add to end of array and then “bubble up”; for dequeue do “trickle down”

**BST (Map)**
- **Structure:** any valid binary tree
- **Order:** leftchild.key $<$ self.key $<$ rightchild.key
  - No duplicate keys
  - Because it’s a Map, values go along for the ride w/keys
- **Big-O:** $\log(n)$ if balanced, but might not be balanced, then $O(n)$
- **Operations:** recursively repeat: start at root and go left if key $<$ root, go right if key $>$ root
Appendix: Sorting
Sorting Tips

Know the differences between algorithms (especially with regard to runtime)

Be able to recognize code for each of the sorting algorithms when you see it

Be able to trace through each sorting algorithm on an initially (unsorted) list
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: $O(N^2)$
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: $O(N^2)$
Heap Sort

Load all the elements into a heap priority queue, then dequeue one-by-one

Runtime: $O(N \log N)$
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 \( \frac{N}{2} \) that we call mergeSort on, then merge the two sorted arrays together

Runtime: \( O(N \log N) \)
Quick Sort

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) $O(N^2)$
Insertion Sort Walkthrough

Insertion Sort walkthrough based on slides by Keith Schwarz
Insertion Sort Walkthrough

Rule: Swap each element to the left until it doesn't have a bigger element before it.
Insertion Sort Walkthrough

Rule: Swap each element to the left until it doesn't have a bigger element before it.
Rule: Swap each element to the left until it doesn't have a bigger element before it.
Insertion Sort Walkthrough

Rule: Swap each element to the left until it doesn't have a bigger element before it.
Rule: Swap each element to the left until it doesn't have a bigger element before it.
Insertion Sort walkthrough based on slides by Keith Schwarz

Rule: Swap each element to the left until it doesn't have a bigger element before it.
Rule: Swap each element to the left until it doesn't have a bigger element before it.
Rule: Swap each element to the left until it doesn't have a bigger element before it.
Rule: Swap each element to the left until it doesn't have a bigger element before it.
Insertion Sort walkthrough based on slides by Keith Schwarz

Rule: Swap each element to the left until it doesn't have a bigger element before it.
Rule: Swap each element to the left until it doesn't have a bigger element before it.
Insertion Sort walkthrough based on slides by Keith Schwarz

Rule: Swap each element to the left until it doesn't have a bigger element before it.
Insertion Sort walkthrough based on slides by Keith Schwarz

Rule: Swap each element to the left until it doesn't have a bigger element before it.
Rule: Swap each element to the left until it doesn't have a bigger element before it.
Rule: Swap each element to the left until it doesn't have a bigger element before it.
Insertion Sort Walkthrough

Rule: Swap each element to the left until it doesn’t have a bigger element before it.
Rule: Swap each element to the left until it doesn't have a bigger element before it.
Rule: Swap each element to the left until it doesn’t have a bigger element before it.
Insertion Sort walkthrough based on slides by Keith Schwarz

Rule: Swap each element to the left until it doesn't have a bigger element before it.
Rule: Swap each element to the left until it doesn't have a bigger element before it.
Rule: Swap each element to the left until it doesn't have a bigger element before it.
Rule: Swap each element to the left until it doesn’t have a bigger element before it.
Insertion Sort Walkthrough

Rule: Swap each element to the left until it doesn't have a bigger element before it.

Insertion Sort walkthrough based on slides by Keith Schwarz