## Course Introduction

Constructing a C++ class
- C++’s use of encapsulation (.h / .cpp files)
- Boilerplate code for C++ classes
- "public" and "private" sections of a C++ class

Building a C++ class
- Constructor

### Pointers

Stack ("automatic") memory in C++
- Stack memory addressing (high addresses, growing down)
- Stack frames
- sizeof() operator

Heap ("allocated") memory in C++
- Heap memory addresses (low addresses, growing up)
- new/delete
- Memory-based operators (unary & and *)

Passing parameters in C++ and tradeoffs
- Pass by value
- Pass by pointer
- Pass by reference
- const modifier

Returns in C++ and tradeoffs:
- Return by value
- Return by pointer
- Return by reference

Operator overloading
Automatic default properties of a class:
- Automatic default constructor
- Automatic default copy constructor
- Automatic default destructor
- Automatic default assignment operator

C++’s "Rule of Three"

### Inheritance

- C++ inheritance syntax (public inheritance)
- Abstract classes in C++

### Templates

- Motivation
- Templated functions
- Templated classes

### List ADT

- Array-based list vs. linked-list list
- C++ Implementation using Templates

### List Analysis by Implementation

- Analysis of insert(), including:
  - Unsorted list, unsorted array: O(1)
  - Sorted array, sorted list: O(n)
- Analysis of insertAfter(*ptr), including:
  - Most notable: Linked list O(1) given pointer
- Analysis of insertAtFront():
  - Most notable: Array amortized O(1) w/ smart resize

### Stack ADT

LIFO ordering property
Analysis: O(1) push() and pop() operations w/ array and list

Array resize strategy: double the size + move the data
Array resize analysis: O(n) operations every O(n) times, amortized O(1)

### Queue ADT

FIFO ordering property

### C++ Iterators:

- Purpose and abstraction
- Use of overloaded operators ++ and *
- Use of ::begin() and ::end()
- Concept of ::end() being "one past the end"

### Functors in C++

- Overloaded call operator, operator()
- Purpose and utility

### Vocabulary:

- vertex/node, edge, path, root, parent, sibling, children, ancestor, descendant, subtree, and leaves
- Recursive definition of a binary tree (not a BST!)
- Tree properties:
  - full binary tree
  - perfect binary tree
<table>
<thead>
<tr>
<th>Tree ADT: insert, remove, and traverse</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tree Proof:</strong> How many NULL points exist in a binary tree with ( n ) nodes?</td>
</tr>
<tr>
<td>Binary tree traversals:</td>
</tr>
<tr>
<td>- in-order</td>
</tr>
<tr>
<td>- pre-order</td>
</tr>
<tr>
<td>- post-order</td>
</tr>
<tr>
<td>- level-order</td>
</tr>
</tbody>
</table>

| Binary tree search:                  |
| - depth-first searching              |
| - breadth-first searching             |

**Understanding the different aims of traversal vs. search**

**Dictionary ADT**

**Binary Search Tree (BST):**
- Recursive ordered property of a BST
- Running times of a BST, in terms of \( n \) and in terms of \( h \)

**Operations on a BST:**
- find()
- Use of return-by-reference to use find for insert() and remove()

**BST Proof:** Minimum number nodes in a tree of height \( h \).
\[ \Rightarrow \text{Largest possible height (} h \text{) given a tree of } n \text{ nodes.} \]

**Comparison of BST best case vs. worst case vs. arrays/lists**

**“Height balance”** \((b)\) of a node (and therefore a tree)

**AVL Tree Rotations:**
- Motivation and purpose
- Four types of rotations: \( L, R, LR, \) and \( RL \)
- Running time of a rotation

**Theorems on which rotation to use based on the height balance**

**Bound on number of rotations:**
- Max 1 rotation on insert
- Max 0 rotations on find
- Max \( \log(n) \) rotations on remove

**AVL Proof:** The maximum height \((h)\) of a tree given \( n \) nodes.
\[ \ldots \text{prove a } 2^\star \log(n) \text{ bound, understand a tighter proof can prove } 1.44. \]

**Applications of AVL:**
- Range-based searching
- Nearest neighbor searching
  - Application: kd-tree

**Motivation of BTree**

Idea: Non-classical analysis of BTree due to not all operations taking the same amount of time

**Understand a BTree of order \( m \) and its properties**

**BTree Operations:** find, insert

**BTree Proof:** Minimum keys on a BTree of order \( m \).

**Motivation of hashing**

**Dictionary ADT w/ a hash table**

**Properties of a hash algorithm:**
- Hash function
  - Properties of a good hash function
  - SUHA
- Array
  - Load factor
  - Running times in term of the load factor
- Collision detection strategy

**Collision detection strategies:**
- Open hashing:
  - Separate Chaining
- Closed hashing:
  - Linear probing
  - Double hashing

**Purpose and utility of hashing vs. balanced BSTs**

**Running times of removeMin() across sorted/unsorted arrays/lists**
\[ \Rightarrow \text{Motivation of a heap data structure} \]

**Recursive definition of a heap**

**Heap operations:** insert, removeMin, buildHeap
- heapifyUp
- heapifyDown
- \( O(n) \) buildHeap

**Applications of heaps:**
- heap sort

**Heap Proof:** Running time of buildHeap is \( O(n) \)

**Motivation of equivalence relations and a disjoint set (representative element)**

**Array-based Disjoint Sets**

**UpTree operations:** union and find
- Lazy union/find
- Smart union: by size, by height
- Path compression

**Running time of an UpTree**
- How does iterated log grow?
- What can we assume about this growth when used in another algorithm?

...and 4 weeks of graphs (covered recently, reviewed in lecture)