CS 24 Notes: Intro to Data Structures

Chapter 1: Phases of Software Development

Specification: precise description of the problem

Order: the run time of an algorithm expressed in Big-O

Big-O:

- constants are ignored
- the dominant term is the asymptotic run time (term with fastest growth rate

Fully Exercising Code: test code that executes every line of code at least once and code that should be skipped should be tested to make sure the code is actually skipped

Chapter 2: Classes and Abstract Data Types

Class Example:

```
class ObjectName {
        private:
                // Member Variables
        public:
                //Member Mutator Functions
                //Const Accessor Functions
};
```

Instance: a variable of object type which keeps its own copies of member variables

Constructor: a member function which is called upon declaration of an instance

- no parenthesis follow an instance if the constructor takes no parameters
- the constructor must have the same name as the class
- does not have a return type and cannot put "void" in the constructor definition
- can have numerous constructors, each taking a different set of arguments

Default Constructor: a constructor with no arguments

Automatic Default Constructor: a constructor which calls only the default constructors of its member variables

Inline Member Function: a function definition inside of the class, which substitutes the function call with said definition of code to enhance run time

Namespace: a name used to localize portions of code to prevent accidental overloading by naming conflicts (defined in both the header and the class implementation file)

Namespace Example:

```
namespace name{
        \\CODE BLOCK
}
```

Global Namespace: items not defined in a specific namespace; can be used without a scope resolution operator

Unnamed Namespace: items that are defined to be local to that namespace

Header File: a file with the extension ".h" that provides class and function definitions, which is enough information a programmer needs to use such class

Header File Comment: comments regarding preconditions, postconditions, and other relevant information a programmer needs to use such a class (inline member definitions do not need to be interpreted)

Macro Guard: guard to prevent header files to be included multiple times (written in the header file)

Macro Guard Example:

```
#ifndef FILENAME_H
#def FILENAME_H
namespace name{
        \\Class Declaration
}
#endif
```

Value Semantics: determines how values of an object is copied to another

Assignment Operator: x = y; copies the values of y to the variable x

Automatic Assignment Operator: the member variables of y are copied to x for its member variables

Copy Constructor: objectType newVar(orgVar); objectType newVar = orgVar; initializes newVar as a copy of orgVar

Automatic Copy Constructor: initializes a new object by copying all the member variables

Implementation File: a file containing function and class implementations which the programmer should not have to understand

- Include all header files using #include "filename.h"

Default Argument: a value used for an argument when the argument is not provided (specified only in the specification file)

Formal Parameter: the parameter of a function

Argument: value passed into a function

Binary Function: function with two arguments

Operator Overloading: defining a new meaning for a defined operator (op) using the "operator" keyword

Operator Overloading Example:

```
returnType operator op(arg1, arg2){
        \\CODE BLOCK
}
```

Friend Function: a function preceded by keyword "friend" declared within a class to give access to its member variables even though the function is not a member function

Chapter 3: Container Classes

Container Class: a class where each object contains a collection of items

std::size t: an unsigned integer type

Static: every instance object of a class uses that same value (the variable is preceded by the "static" keyword)

- allows use of the scope resolution operator to determine the value since it is the same for all objects
- static should not be declared in an implementation file

Invariant of the Class: rules that dictate how the member variables of a class represent a value or object

Assertions: use public member functions instead of private member variables to maintain concept of abstraction

Chapter 4: Pointers and Dynamic Arrays

bad_alloc Exception: exception that arises from failure to allocate memory using the keyword "new"

Arrays: an array variable is a pointer to the first element of the array

Const Pointers: a pointer preceded by the keyword "const" means the value the pointer points to cannot be changed by dereferencing the pointer

Dynamic Data Structure: a data structure whose size is determined while running and not compile time

Deep Copy: making a copy of dynamic variables using the non-default constructors/operators because the object should be copied, not only the memory address

Dynamic Classes: a classes that uses dynamic memory

- automatic operators, copy constructor, and destructor need to be overridden

Return Location: the copy of the local variable when the function returns and the local variable is destroyed

Destructor: a member function which is called to deallocate heap memory

- no parenthesis follow an instance if the constructor takes no parameters
- the constructor must have the same name as the class preceded by a tilde
- does not have a return type and cannot put "void" in the constructor definition
- rarely called explicitly since automatically called when objects become inaccessible

Chapter 5: Linked Lists

- return types can be const to prevent accidental pointer manipulation errors
- arrays are better at random access; linked lists are better with inserting and deleting at the head

Chapter 6: Templates and Iterators

Templates: functions and classes which can be used with different data types

Iterator: an item which iterates through all items in a container

Item: a name of the underlying data type of a template; when a template function or class is used, the compiler will determine the type of Item; an Item is an example of a template parameter

- Non template parameters must match exact type (const size t vs int vs size t)

Template Prefix: "template <class Item>" tells the compiler that the following definition uses an unspecified data type

Template Function Example:

```
template <class Item>
returnType functionName(args){
        //CODE block;
}
```

Instantiation: An instantiation of a template function or class is a use of the template function with a specific data type

Unification Error: failure for compiler to determine how to instantiate a template function or class; template parameters must appear in the parameter list of the template function

Template Classes:

- must still have template prefix for each function that uses template parameters/return types
- must be implemented in the header file; (include the implementation at the end of header instead of top of implementation file)
- must use keyword "typename" outside of member functions and class definition to reference the class's types
    - Ex: typeName className<InstantiatedType>::type

Output Iterators: an iterator to put elements into a result container

Input Iterators: an iterator to be dereferenced and access elements

Forward Iterator: must satisfy the following properties

- has default constructor, copy constructor, and assignment operator
- can act as input iterator
- ++ operator can iterate the iterator to the next item
- can be tested for equality (if current items are in the same position, then true)

Bidirectional Iterator: a forward iterator which can move backward with the -- operator

Random Access: quickly access a randomly selected location in a container

Random Access Iterator: has additional operators that work in addition to the bidirectional iterator, including p[n] accesses the nth element ahead of the current element (different from array iterators)

Reference Return Types: prevents returning of local variables; function returns the actual variable or object instead of a copy of the objects (assignment can be made to the function return value now)

- assigning the value to a variable will still make a copy to the new variable
- cannot be used as a constant member function

Note: prefix ++ operator is more efficient than postfix ++ operator since the ladder makes a copy even if the return value is never used

Chapter 7: Stacks

Stacks: data structure with ordered entries such that the entries can be inserted and removed only at one end (the top)

Last In First Out: Entries are taken out of the stack in the reverse order of their insertion

Stack Underflow Error: Pop Empty Stack

Stack Overflow Error: Push onto a Full Stack

Koenig Lookup: use of arguments to determine which functions to use

Infix Notation: Arithmetic Operator in between Operands (Ex. 2 + 3 = 5)

(Polish) Prefix Notation: Arithmetic Operator Precedes Operands (Ex. + 2 3 = 5)

(Polish) Postfix Notation: Arithmetic Operator Proceeds Operands (Ex. 2 3 + = 5)

Chapter 8: Queues

Queue: data structure with ordered entries such that entries can only be inserted at one end (the rear) and removed at the other end (the front).

First in First Out: Entries are taken out of the queue in the same order as insertion

Queue Overflow: pushing into full queue

Queue Underflow: popping out of empty queue

Circular Array: array that connects last element to first by using modulo

Deque: Double-ended queue in which entries can be inserted and removed from both ends

Chapter 9: Recursion

Activation Record: place where function execution is stored while a different function is called and until that is returned since function stops execution until the inside function is finished executing

Run Time Stack: collection of activation records

Fractal: an object which looks the same after magnified

Chapter 10: Trees

Tree: has a finite set of nodes

- one special node: the root
- each node may have 1 or more children
- each node must have 1 parent unless it's the root
- by moving upward, you will eventually reach the root

Empty Tree: a tree with zero nodes

Node: consists of a datum, left child, and a right child

Root: the top node of a tree

Left Child: the node linked on the left

Right Child: the node linked on the right

Leaf: node with no children

Parent: the node in which a child was linked from

Siblings: two nodes are siblings if they come from the same parent

Ancestor: A node's parent (or parent of a node's parents, etc..)

Descendant: A child of a node (or child of a node's child, etc...)

Left Subtree: a tree beginning with a node's left child

Right Subtree: a tree beginning with a node's right child

Depth of a Node: the number of node it takes to reach the root (with the root having a depth of 0)

Depth of a Tree: maximum number of nodes it takes to reach the root

Full Binary Tree: every leaf has the same depth and every non leaf has 2 children

Complete Binary Tree: a full binary tree with added nodes that are as left most as possible

A binary tree can be represented using an array with the following formulas:

- Parent = floor((i-1)/2)
- Left Child = 2i + 1
- Right Child = 2i + 2

Preorder Traversal: root is processed prior to processing the left, then the right subtree

Inorder Traversal: left subtree is processed, then the root, then the right subtree

Postorder Traversal: left, then right subtrees, then the root is processed

Functions can be passed as parameters:

Ex: returnType functionName(returnTypeOfFunctionParameter functionParemeter(itsParameters), dataType parameter)

Binary Search Trees: the datum of a node is never less than one of the datums in its left subtree and never greater than one of the datums in its right subtree

Chapter 11: Balanced Trees

Heaps: binary tree with an overloaded less-than operator for comparison

- max-heaps have parents greater than their children
- trees are a complete binary tree

Priority Queue: queue with relative priorities of each item in the queue

- implemented using a max heap

B-Trees: tree to prevent linked list configuration

- root can have as few as one entry
- every other node will have at least a MINIMUM number of entries and at most double the MINIMUM
- each node stored in a partially filled, sorted array
- number of subtrees is 1 more than the number of entries in the node
- an entry at index i of a parent node is greater than an entries in the ith subtree; an entry at index i of a parent node is greater than an entries in the i+1th subtree
- every leaf has the same depth

Chapter 12: Searching

Serial Search: stepping through an array sequentially to find an element

- Worst Case: O(n)
- Average Case: O(n)
- Best Case: O(1)

Binary Search: stepping through a sorted array comparing the median

- Worst Case: O(log n)
- Average Case: O(log n)
- Best Case: O(1)

Depth of Recursive Calls: longest chain of possible recursive calls

Hashing: a hashing function transforms a key into a hash location where the element would be stored if it exists

Simple Hashes include...

- **Division Hash:** key % capacity
- **Mid Square Hash:** middle digits of key * key
- **Multiplicative Hash:** first few digits of key * fractional constant

Hash Collision: if two elements hash to the same location

Open-Address (Linear Probing) Hashing: in the event of a collision, an element would be inserted at the next available location

- **Clustering:** when a group of keys are hashed to the same location, resulting in a cluster in the table

Double Hashing: in the event of a collision, jump a number of spots equal to a second hash

- reduces clustering

Chained Hashing: each component of the table is a linked list / vector structure that can dynamically add and remove elements

Load Factor: elements in table / size of table

Chapter 13: Sorting

Selection Sort: selects the highest element in an array and swaps it with the last element; repeats for next highest until the array is sorted

Insertion Sort: starts with the first element and inserts into a sorted list; then adds second element to sorted list and repeats until all elements are inserted into the sorted list

Divide and Conquer: divide a problem into smaller subproblems, solve each subproblem and use those solutions to solve the problem

Merge Sort: divide a list into two smaller sublists; repeat until one element (sorted) lists; then, merge each sorted list into a bigger list

Quick Sort: partition a list into two sublists, one greater and smaller than a pivot; recursively sort each sublist until the sublists become 0 or 1 element (sorted)

Heap: a complete binary tree with elements that can be compared with the less than operator and has a strict weak ordering

- root at 0th index of array
- left child index = 2 * parent index + 1
- right child index = 2 * parent index + 2

Heapsort:

1. Create the heap: O(n log n): if parent is smaller than either child, swap parent with largest child and trickle
2. Trickle: O(n log n): swap element at top of heap (largest element), with the last element in the array; then trickle downwards.

Chapter 14: Derived Classes and Inheritance

Pure Virtual Functions: denoted by = 0; as a part of the function declaration

Pure Virtual Function example:

returnType functionName(paramTypes paramVars) = 0;

Abstract Classes: classes containing only pure virtual functions and cannot be instantiated

Multiple Inheritance: a single derived class and inherit from multiple base classes

Chapter 15: Graphs

Undirected Graph: set of finite nodes and finite links connecting each node

- **Vertex:** node
- **Edge:** links two vertices

Directed Graph: each edge connects a source to a target

- **Loop:** connects vertex to itself
- **Path:** a sequence of vertices connected from source to target to new target to new target...
- **Multiple Edges:** more than one edge connected in the same direction

- **Simple Graph:** no loops nor multiple edges

Adjacency Matrix: square grid of true/false values where true at row i column j implies there is an edge from vertex i to vertex j

Breadth-First Search: searching all adjacent edges before continuing deeper

Depth-First Search: going as deep as possible along a path before trying a different path