CS 32 Study Guide: Algorithms, Data Structure vcs, Abstract Data Types, Headers, Linked Lists, Stacks, Queues, Maps, Inheritance

An algorithm is a set of instructions/steps that solve a particular problem.
The imporance of algorithms is: RUNTIME

A data structure the data that's ope ated on by an algorithm to solve a problem.

Abstract Data Type (ADT):
The collection of (a) data structures, (b) algorithms and (c) interface required to solve a particular problem. The ADT provides an interface to secret algorithms and data structures In C++, ADT's are defined as Classes

Object Oriented Programming:
programs are co structed from multiple self-contained classes.

Examples of Algorithms:
-Linear search
-Binary search

```
/* NEVER INCLUDE A .CPP FILE
IN ANOTHER FILE. ONLY
INCLUDE .H FILES
NEVER PUT 'USING NAMESPACE
STD' IN A HEADER*/
```

Preprocessor Directives:

```
#ifdef FILE_H
 //checks if already defined
#ifndef FILE_H
 //checks if not defined
#define FILE_H
 //defines a constant
#endif //like an end bracket
```

```
01101010
01001100
01001011
10110101
```

```
/* use include guards
to prevent multiple
definitions */
```

constructors/destructors

```
/*if you declare an array of objects,
that object must have a default
constructor that requires no arguments*/
Class csNerd
{
  public:
    csNerd(int PCs, bool UsesMac)
      :m_numPCs(PCs), m_MacUser(UsesMac)
      //initializer list
    {…}
    ~csNerd(); //destructor, only one!
}
/*desctructors must: Free any dynamically
allocated memory, close any opened disk
files, and disconnect any opened network
connections*/
```

```
/* Class co position: If
a class contains one or
more classes as member
variables, */
```

```
/*include header files
when you define a
variable of that class
type or call any member
function from that
class.
DO NOT include header
files if you define a
parameter, return type
or pointer/reference
variable of the class */
class csNerd;//instead
```

Copying Stuff

```
Class Circ{
  public:
    Circ();
    Circ(const Circ& old);
      //copy constructor
    Circ& operator=(const Circ& source)
      //assignment operator
    {...
      return (*this); //required!
    }
}
int main(){
  circ one;
  circ two;
  two = one; //assignment operator call
  circ three(two); //copy constructor
}
```

```
/*a default copy constructor performs a shallow copy,
which does not work on dynamically allocated data or
opened system resources.
A copy constructor must:
-  determine how much memory is allocated by the old
   variable
-  allocate the same amount of memory in the new
   variable
-  copy the contents*/
```

```
/* the default assignment operator performs a shallow
copy, while will not work on dynamically allocated data
or any system resources that have been opened.
A assignment operator must:
-  free all dynamic memory used by the target instance
-  Re-allocate memory in the target instance to hold any
   member variables from the source instance
-  explicitly copy the contents of the source instance
   to the target instance*/
```

```
class Stack{
public:
  stack(); //constructor
  void push(int i); //add to stack
  int pop(); //remove from stack
  bool is_empty(void);
  int peek_top(); //return top value
  …
}
```

```
class Queue{
public:
  enqueue(int a); //adds a to end
  int dequeue(); //removes first
  bool isEmpty();
  int size();
  int getFront() //get front value
}
```

Linked Lists: (doubly linked)

```
struct node
{
  string name;
  node* next;
  node* prev;
}

class myLinkedList
{
public:
  void addtoFront(string name);
  void deleteItem(string name);
  void deleteItem(int slotNum);
  int find(string name);
  void print();
  myLinkedList() //creates empty list
  { first = last = NULL }
  ~myLinked List();
private:
  node* first //beg of list
  node* last  //end of list
```

```
/* You can create linked
lists that are singly linked,
doubly linked, or in a loop
depending on what you need */
```

CHECK THE BOUNDARY CONDITIONS

```
/*inert algrithms that insert
at the top are the easiest to
code and the fastest.
Middle/end are slower/more
complex*/
```

```
/* Destructors must traverse
the entire linked list */
```

Destructing a Derived type

1. Execute the body of the destructor
2. Destroy data members
3. Destroy base part

Linked List Vs. Array
Array is Faster for
- getting a specific item
- less debugging problems

Linked List is Faster for
- inserting at the front
  removing from the middle

Circular Queue: use pointers head and tail to loop around an array

MAKE SURE THE POINTER DOESN'T POINT TO NULL

Constructing a Derived Type

1. Construct base part
2. Construct data members
3. Execut the body of the constructor

```
/* Derived classes can
only access public member
variables and functions of
the base class If you want
Derived classes, but not
the public to access
variables, use protected*/
```


```
/* Copy Constructors and assignment
operators will copy the base and
derived data correctly, UNLESS it is
dynamically allo aited */
```

RECURSION:

1. Identify if the problem is repetitive on a broad scale and/or can be simplified
2. Identify the simplist, complete case
3. Identify the base cases

```
if(base case)
    dosomething
else
    dosomething to reduce the size of
    the problem
```

Inheritance

```
class Base
{
  public
    Base(int p1, int p2)
    void doThis(); //!!!!!
    virtual void doIf(); //default: derived, if it exists
    virtual void doIf2() const =0; //pure virtual
  private:
    [stuff…]
}
class Derived : Public Base
{
  public
    Derived(int p1, int p2) : Base(p1, p2) {}
        //base must be constructed, or default is used
    virtual void doIf2() const;
        //declare overrides virtual as well
    virtual void doIf();
}

void Derived::doIf()
{    Base::doIf2();   }
//to call in a derived class a function from the base
//class that has been overwritten, you need to use
//'Base::'
```

```
/* Recursive functions
should never use
global, static, or
member variables, only
local variables and
parameters! */
```

Generic Programming:
override/define generic comparison operators (<, >, ==, etc)
then, use templates! ☺

TEMPLATE CODE:

```
template <typename T>
   //indicates the following class
   //or function is a template
void function(T a[], T p2)
   //T type must be passed as a
   //parameter!
{
     T total = T(); //see*
     ...
}

void function(int a[], int p2)
{...}  //you can write exceptions the
       //compiler will default to

template <typename T1, T2>
    //multi-type templates work too!
void f2(T1 a[], T2 b[])
```

```
/* In templates, the compiler uses
template argument deduction (checks the
parameters) to figure out what functions
to use. Non-template matches have
priority, then te plate matches. If the
call does not match the template exactly,
there will be a compile time error!*/
```

```
/* Using the term T()
allows you to
initialize to the
"default constructor"
of whatever type you
use. For numbers,
this is 0. Bools are
false, strings are
empty, chars are the
0 byte. */
```

ALWAYS PLACE TEMPLATES IN THE HEADER FILE

```
/* when you have a
   function that
traverses the entire
leftover list each
time, the algorithm
has time complexity
O(N^2): N(N+1)/2 =
1/2N^2+1/2N)*/
```

Template Classes

```
template <typename T>
class something
{…};

template <typename T>
void something<T>::f1(T a)
{…};
```

Inline Functions:

```
/* anything declared inside the class
declaration is automatically inline: the
compiler copies the code wherever you
call the function, speeding up the
program because there's less jumping.
declare external functions inline like
this: */

inline void sclass::f1()
{}

/* setting large functions inline will
greatly increase your exe file size
*/
```

Runtime Time Complexity

```
/*written in terms of "Big
'O' Notation" O(some
function of N), where N is
the number of data terms.
Things to consider if
complexity varies:
Best Case Time
Worst Case Time
Average Case Time
Does your data cause you to
generate the Best/Worst case
often? */
```

```
/* sometimes, for things
like sorting, you consider
complexity of swaps over
comparisons (or some other
specific action) because it
takes significantly longer.
Usually, the longer one is
not swaps, because you
should SWAP POINTERS */
```

INFIX TO POSTFIX

```
Initialize postfix to null
Initialize the operator stack to empty
For each character ch in the infix string
    Switch (ch)
        case operand:
           append ch to end of postfix
           break
        case '(':
           push ch onto the operator stack
           break
        case ')':
             // pop stack until matching '('
           While stack top is not '('
             append the stack top to postfix
             pop the stack
           pop the stack  // remove the '('
           break
        case operator:
           while the stack is not empty and the stack top is not '('
            and precedence(ch) <= precedence(stack top)
                append the stack top to postfix
                 pop the stack
           push ch onto the stack
           break
While the stack is not empty
    append the stack top to postfix
    pop the stack
```

Evaluating Postfix

```
Initialize the operand stack to empty
    For each character ch in the postfix string
        if ch is an operand
            push the value that ch represents onto the operand stack
        else // ch is an operator
            set operand2 to the top of the operand stack
            pop the stack
             set operand1 to the top of the operand stack
             pop the stack
             apply the operation that ch represents to operand1 and operand2,
                and push the result onto the stack
    When the loop is finished, the operand stack will contain one item,
      the result of evaluating the expression
```

Passing functions as parameters to functions:

```
double g(int x);
double integrate(int xlow, int xhigh, double f(int))
{
      double y= (*f)(x) //or f(x);
}

main()
{
      double area = integrate(low, high, g);
}
```