Study Guide 1: COMP15

Pointer Review

- stored in the stack, point to the heap or stack

```
//POINTERS
int c = 10;
int &p = NULL;
p = &c; // & = referencing pointer
int *q = NULL;
int q2 = 2;
q = &q2; // dereference once again
p = q;// set pointers equal to each other
//DECLARING AN ARRAY OF POINTERS
int *arr_ptr = new int[5];
int **arr_ptr_2 = new int[3][3];
//DELETE ARRAYS
delete []arr_ptr; //delete 1d array
delete [][]arr_ptr_2; //delete 2d array
```

Leak Check

- VALGRIND: check for leak: ***valgrind –leak -ch=full ./a.out***
- memory errors could occur such as in the case of double free, which would return an invalid free error

GDB & Debugging

- formalize expectations for expected use-cases
- robustify code
- can write test cases for functions
- two testing paradigms for testing:
  - exhaustive: test every possible input
  - random: test a subset of possible inputs to act w/ a degree of probability
- to run: ***gdb ./program_name***
- then: ***run***
- running **where** shows when the program's stack calss upon exit
- use ***q*** to quit gdb
- to silence unused variable warnings: typecast the variable as void:
  - ***(void) argc***
  - ***(void) argv***

Difference Between Heap and Stack

- **STACK:**
  - memory set aside as scratch space for thread of execution
  - when a function is called, a block is reserved on top of the stack for local vars
  - when that function returns, the block becomes unused and can be used the

next time the function is called
    - the stack is always resered in a LIFO order (most recent reserved block is always the next to be freed)
    - makes it simple to keep track of the stack
    - to free a block from the stack, you can just adjust one pointer
- **HEAP:**
    - memory set aside for dynamic allocation
    - no enforced pattern for allocation and deallocation of blocks from heap
    - can allocate a block and free it at any time
    - makes it more complex to keep track of which parts of the heap are allocated or free at any given time
- **BOTH:**
    - each thread gets a stack, whereas there is usually only one heap for the program
    - there can be multiple heaps for different types of allocation
    - scope of a stack depends on the thread it is attached to; when the thread exits, the stack is reclaimed
    - the heap is allocated at runtime, and is reclaimed when the application exits
    - the size of the stack is set when a thread is created
    - the size of the heap can grow as needed
    - stack is faster because access pattern is trivial to allocate and deallocate (a pointer is simply incremented/decremented)
    - the heap is far more complex due to the bookkeeping involved in allocation/deallocation
    - bytes in stacks are mapped to the processors cache, so that makes it faster to
    - the heap is a global resoruce and is slower
- **FURTHERMORE:**
    - **STACK:**
    - Stored in computer RAM just like the heap.
    - Variables created on the stack will go out of scope and are automatically deallocated.
    - Much faster to allocate in comparison to variables on the heap.
    - Implemented with an actual stack data structure.
    - Stores local data, return addresses, used for parameter passing.
    - Can have a stack overflow when too much of the stack is used (mostly from infinite or too deep recursion, very large allocations).
    - Data created on the stack can be used without pointers.
    - You would use the stack if you know exactly how much data you need to allocate before compile time and it is not too big.
    - Usually has a maximum size already determined when your program starts.
    - **HEAP:**
    - Stored in computer RAM just like the stack.
    - In C++, variables on the heap must be destroyed manually and never fall out of scope. The data is freed with delete, delete[], or free.
    - Slower to allocate in comparison to variables on the stack.
    - Used on demand to allocate a block of data for use by the program.
    - Can have fragmentation when there are a lot of allocations and deallocations.
    - In C++ or C, data created on the heap will be pointed to by pointers and allocated with new or malloc respectively.

- Can have allocation failures if too big of a buffer is requested to be allocated.
- You would use the heap if you don't know exactly how much data you will need at run time or if you need to allocate a lot of data.
- Responsible for memory leaks.

Runtime Complexities

\- **Runtime Complexity of Expand Function on ArrayList:**
\- call expand() function when the array reaches capacity
\- takes 28 steps to insert 16 items (better than ~256 it would take if one had expanded every time)
\- amortized O(1)

Stacks

- LIFO
- only two operations: pop and push
- recursive data structure
- all stack operations must run in O(1)
- LL implementation of Stack is most efficient in making it dynamic

Queues

- contianer of objects that inserted according to FIFO
- only two operations allowed: enqueue and dequeue
- enqueue: to insert an item into the back of the queue
- dequeue: to remove the front item

Differences between Stacks and Queues:

- removal
- each operation must run in O(1)

ArrayLists

- contiguous in memory
- size can change dynamically; use a dynamically allocated array to store elements
- ArrayLists consume more memory in exchange fro the ability to manage storage and grow dynamically in an efficient way
- allocated memory for its elements on the heap
- frees heap memory when it goes out of scope (calls destructor)
- accessing any element is O(1) due to it being contiguous in memory
- inserting elements at the tail is O(1), but inserting at the head is O(n)

LinkedLIsts

- uses dynamic memory allocation (stored on heap)
- made up of node objects, which have a pointer to the next node object
- implementing header node eliminates special cases:

- to insert a new element:
  - first new list node created
  - then value is placed in curr->next
  - once node is added, tail is pushed forwared
  - insertion is O(1)

Comparison of Both

- ArrayLists have disadvantage that size must be predetermined before array is allocated.
- Cannot grow beyond predetermined size
- Link lists have advantage that they only need space for objects located on their list
- ArrayLists: advantage that there is no wasted space for an individual element
- LinkedLists: requre an additional pointer for ever list node
- amount of space required for an ArrayList: DE(size of data elemnt * max. number of list elements)
- for LinkedList: n(P+E), P is size of pointer in storage, n is number of elements on list
- LinkedList are more space efficient when number of elements varies or is unknown
- ArrayLists are space efficient when the max lenght is known
- O(n) to access element in LL, O(1) in AL
- O(1) for insert and removal in LL, O(n) in AL

Implementations

- **see appendix a.(a) for code examples**

The Rule of Three

- for any class with dynamic memory, we write a copy constructor, assignment operator, and destructor

Copy Constructor and Assignment Operator

- copy constructor invoked when we're making a new object
- assignment operator invoked when we replace an existing object

```
LL myList; #invokes default constructor
#myList is pushed on to stack
#inserting into LL allocates nodes on the heap

LL l2; #calls default constructor (on execution stack)

l2 = myList #replacing calls assignment operator
#not using assignment operator points l2 and myList
#to the same location in memory in the heap, so destructing one of them destructs both of them. cal

LL l3 = foo(myList) #invokes copy constructor, scope of copy constructor is in foo, and somewhere i
```

- **see appendix a.(b) for code examples**

Binary Trees

- every node has 0, 1, or 2 children
- a leaf is a node with no children
- root is at the top of the tree (entry point)
- a node has a **depth** (number of levels away from root)
- a tree's height is the depth of the deepest node
- **Binary Search Tree**:
  - is a type of binary tree
  - no duplicates
  - can be unbalanced
  - but nicely sorted
  - anode is smaller than right subtree, but larger than left subtree
- BinaryNode struct:

```
class BinaryTree //base class
{
    public:

    protected; //instead of private
        BinaryNode *root;
};

class BST: public BT //derivedclass
{
    public:
};

struct BinaryNode
{
    int value;
    BinaryNode *left;
    BInaryNode *right;
}//end struct
```

Traversals + Recursion

- Functions the same for BT and BST
- structure, not values
- traversal 'visit' every node
- print every node of the tree
- recursive functions b/c a subtree is a binary tree
- base case (usally empty tree)
- recursive step (go left, or right)
- Pseudocode:

```
Pre-order traversal pseudocode:
    -print root
    -recursively print left subtree
    - recurseivley print right subtree

void print_pre(BinaryNode *tree, ostream &out)
```

```
{
    if (tree == NULL)
        {
            return;
        {
        else
        {
            out << tree->value << endl;
            print_pre(BinaryNode tree->left, out);
            print_pre(BinaryNode tree->right, out);
        }
}//end print_pre
```

BinaryTree and BinarySearchTree

- depth = height - 1
- a leaf is a node with no children
- a full binary tree has leaves or nodes with two children
- in a complete binary tree all levels besides possible d-1 is full
- the number of leaves in a non-empty full binary tree is one more tahn the number of internal nodes

Traversals

- Preorder: Left, Root, Right
- Postorder: Root, Left, Right
- Inorder: Left, Right, Root

Functions not Inherited in BST

- destructor
- CC and AO
- insert
- find
- remove: http://www.algolist.net/Data_structures/Binary_search_tree/Removal

- **see appendix a.(c) for code sample**

Inheritance

- polymorphism allows for less code reuse
- **polymorphism** - ability to treat objects of different data types as if they were the same
- protected data members can only be accessible from derived classes
- virtual member functions can be overrided by the dervied class, enabling polymorphism
- virtual function is late binding ; BinaryTree pointer is compile time(early), new keyword is run-time (late)
- no such thing as a virtual constructor
- late binding: runtime

- early binding: compile time
- AVL Tree is a Binary Tree
- a Red Black tree is a BST
- in .h file of derived class: **class BinarySearchTree : public BinaryTree**
- constructors are not inherited and cannot be virtual
- both constructors are written and called, base class is called first, derived class is called second
- destructors can be inherited and C++ will insist that the base class destructor is virtual, and both are called when the object goes out of scope
- BST destructor first, BT destructor second (derived, then base)