**CS 138 Midterm Review**

**Background:**

* **OS & Shells**
* Unix history:
  + Unix/C were developed together in 1970s @ Bell Labs/AT&T
  + Richard Stallman founded the GNU project in 1984 – a “truly free” operating system
  + Linux – open source, just the OS kernel
* Unix command line:
  + A Unix bash command:
    - $ cmd –opt1 –opt2 arg1 –opt3 arg2 arg3
  + Alias: e.g. rm → rm –iv
  + $ which cmd / $ type cmd
    - Is cmd an alias, built-in, an executable, etc.
  + $ man cmd
    - Prints information about command
  + $ ls
    - Lists contents of directory
    - -l gives long listing (shows file permissions)
  + $ mkdir dir-name
    - Creates new directory
  + $ cp src-file tgt-file / $ cp src-file tgt-dir
    - Copies file in current or different directory
  + $ mv src-file tgt-file / $ mv src-file tgt-dir
    - Renames file in current directory or moves it to different directory
  + $ rm file-or-dirlist
    - Delete files
  + $ cat file
    - Prints file in one continuous stream
  + $ less file
    - Prints file one page at a time
* File permissions
  + Access levels: user, group, other
  + Permission: read, write, execute
    - For directories – read = ls; write = add/delete files; execute = cd
  + - --- --- --- → file/directory, user permissions, group permissions, other permissions
    - E.g. - rwx r-x ---
  + $ chgrp grp-name file-or-dir
    - Change group name associated with file(s)
  + $ chmod mode-list file-or-dir
    - Change permissions of file(s)
    - mode-list = level operator permission
    - E.g. $ chmod u-r g+w o-x file
* **Programming languages timeline**
  + 1940-1950s – machine language (punch cards)
  + 50s – first high level language (FORTRAN)
  + 60s – first OO languages
  + 70s – BASIC
  + 80s – C++
  + 90s – Java
  + Unix, GNU

**Basics:**

* **Strings**
  + + → concatenation
    - String literals are char\* and cannot be concatenated (cast to string)
  + ==, <=, >= → lexicographical comparison
  + s.length() → length
  + s.substr(i, n) → n chars startin at i
  + s.c\_str() → returns char\* of string
  + s.find(str, i) → 1st index of str in s starting at i
  + s.clear() → empty string
* **Input/Output**
* #include<iostream>
  + cin >> s → read token
    - while (cin >> nextToken) { … }
  + getline (cin, s); → read line of input
  + cout << s << endl; → output
  + cerr << s << endl; → output error message
* #include<fstream>
  + ifstream input;

input.open (“inputFile.txt”);

while (input >> s) { … }

input.close();

* + ofstream output;

output.open (“outputFile.txt”);

output << s << endl;

output.close();

* **Memory**
* Run-time stack
  + Local variables & parameters
  + Direct instantiation:
  + someStruct s;

s.field; s.method();

* + Instance destroyed at the end of defining scope
* Heap/freestore
  + Access by pointers only
  + Allocate with new, deallocate with delete or delete[] (contiguous)
  + Dynamic instantiation:
  + someStruct \*s = new someStruct;

s->field; s->method();

delete s;

* **Arrays/Vectors**
* Both contiguous memory
* Array
  + string s[N];
    - Static (on stack); array bound must be a constant
  + string \*s = new string[n]; delete [] s;
    - Dynamic (on heap); array bound can be a variable
* Vector – #include<vector>
  + vector<int> v; → both dynamic (on heap)
  + vector<int> v (10);
  + Size <= capacity
* Stack as a vector:
  + v.push\_back(e); v.pop\_back(e); v.back();
* s.at(n) → checked access
* s[n] → unchecked access

**Concepts:**

* **Defensive programming**
* Use a debugger
* Insert print statements
  + At beginning/end of function call
  + Before deleting something
  + Print the whole data structure
* Insert assertions – #include<cassert>
  + To check the logic/assumptions are correct
  + To check the user is being reasonable
* Regression testing – retesting after changes are made
* Test-driven development – first develop test cases that satisfy the requirements
  + Then write the simplest code that can pass the tests
* Blackbox testing – only testing against specifications
* Whitebox testing – testing against implementation (code)
* **Abstract data types**
* Contains data that can only be accessed via a set of operations
  + Signature – describes parameters & return type
  + Pre-condition – assumed to be true before the operation is applied
  + Post-condition – value/effect of the operation
* Multiple ways to implement an ADT, but its abstract specifications are the same
* An interface is created around an ADT – enforces limited access to internal details
* Adapter – an interface that provides exactly what the client needs and no more
  + Prevents client from depending on features they shouldn’t use
  + Protects them from changes in underlying implementation
* Information hiding – separating information from implementation
* **Reference parameters**
* Call by value – C/Java
  + Copy of parameter is created on the call stack
  + Changes made to parameter values do not propagate back to calling environment
* Call by reference – C++
  + The reference param is an alias for the variable in calling environment
  + Changes made do propagate back to caller
  + Const reference param is not allowed to be changed
  + void function1 (Object x);
    - Copy of x created on call stack, changes to x do not propagate back
  + void function2 (const Object x);
    - Copy of x created on call stack, x may not be changed
  + void function3 (Object & x);
    - x refers to variable in calling environment, changes propagate back
  + void function4 (const Object & x);
    - x refers to variable in calling environment, may not be changed
* **Scope**
* Global variable – defined outside of any enclosing function/class
  + Declared →→ end of program
* Local variable (params) – defined within a function body
  + Function called →→ function returns
* Member/instance variable – part of an instance of a struct/class
  + Instance created →→ instance destroyed
* Scope – extent to which an identifier is visible
  + An activation record/stack frame is created for each new scope entered, destroyed when leaving scope
* **Recursion**
* E.g. factorial
  + Base case(s) – if (n <= 1) return 1;
  + Reduction operator – factorial(n-1);
  + Composition operator – return n \* factorial(n-1);

**Data Structures:**

* **Stack**
  + void initStack(Stack & s);
    - Nothing → an empty stack
  + bool isEmpty(const Stack & s);
    - Stack → same stack (returns if size == 0)
  + void push(Stack & s, string val);
    - Stack → stack with new element inserted in front
  + void pop(Stack & s);
    - Non-empty stack → stack with first element removed
  + string peek(const Stack & s);
    - Non-empty stack → same stack (returns value of first element)
  + void nuke(Stack & s);
    - Stack → nullptr
  + Stack as a linked list:
  + struct Node {

string val;

Node\* next;

};

* + typedef Node\* Stack;
  + Stack as a vector: see above
* **Queue**
  + void initQueue(Queue & q);
    - Nothing → a new queue
  + bool isEmpty(const Queue & q);
    - Queue → same queue (returns if size == 0)
  + void enter(Queue & q, string val);
    - Queue → queue with new element added at the end
  + void leave(Queue & q);
    - Non-empty queue → queue with first element removed
  + string first(const Queue & q);
    - Non-empty queue → same queue (returns value of first element)
  + void nuke(Queue & q);
    - Queue → nullptr;
  + Queue as a vector:
    - enter() calls push\_back()
    - Must keep track of location of first element
    - In a linear array, the queue shifts and leaves wasted memory
  + Queue as a linked list:
  + struct Node {

string val;

Node\* next;

};

* + - struct Queue {

Node\* first;

Node\* last;

};

* **Doubly linked list**
  + struct Node {

string val;

Node\* prev;

Node\* next;

};

* **Sorted linked list**
  + bool lookup(const SortedList & first, string val);
    - Sorted list → same list (returns whether val has been found)
    - Cases: list is empty / list is not empty
  + void insert(SortedList & first, string val);
    - Sorted list → sorted list with new element
    - Cases: val is first element / val is after first element
  + void remove(SortedList & first, string val);
    - Non-empty sorted list → sorted list with given element removed
    - Cases: val is first element / val is after first element
* **Priority queue**
  + Either highest or lowest priority means “most important”
  + void enter(PQ & pq, string val, int priority);
    - Priority queue → priority queue with new element added at the end of its priority
  + void leave(PQ & pq);
    - Non-empty priority queue → priority queue with first element of the highest priority removed
  + string first(const PQ & pq);
    - Non-empty priority queue → same PQ (returns first element of the highest priority)
  + Priority queue as a list of lists:
  + struct QNode {

string val;

QNode\* next;

}

* + struct PQNode {

int priority;

QNode\* first;

QNode\* last;

PQNode\* next;

};

* + typedef \*PQNode PQ;
  + Priority queue as a heap:
    - Heap property – value of parent >= value of children
    - Largest element is always at root node
* **Binary search tree**
  + Each node has 0 to 2 children & 1 parent
    - Root node has no parent
  + Leaf – node w/ no children; internal node – anyone else
  + Height – longest path from node to a leaf (down)
  + Depth – path from root to node (up)
  + BST property – value of all nodes in left/right subtree </> value of node
  + struct QNode {

string key;

Node\* left;

Node\* right;

}

* + typedef \*Node BST;
  + bool lookup(const BST & root, string key);
    - BST → same BST (returns whether given element was found)
    - Recursively call lookup for left & right subtrees
  + void print(const BST & root);
    - In-order traversal – print left subtree, current node, then right subtree
  + void insert(BST & root, string key);
    - BST → BST with new element added
    - Navigate recursively until at leaf, add new element as a child of leaf
  + void delete(BST & root, string key);
    - Non-empty BST → BST with element removed
    - Cases: node has no children / node has 1 child / node has 2 children
      * If node has 2 children, “replacement node” is the node w/ max value in the left subtree
* **Time complexities**
* Queue (linked list)
  + Enter = O(1)
  + Leave = O(1)
  + Search = O(n)
* Stack (linked list)
  + Push = O(1)
  + Pop = O(1)
  + Peek = O(1)
  + Search = O(n)
* Sorted linked list
  + Insert = O(n)
  + Delete = O(n)
  + Search = O(n)
* Sorted array
  + Insert = O(n)
  + Delete = O(log n)
  + Search = O(log n)
* Priority queue (list of lists)
  + Enter = O(k) → # of priorities
  + Leave = O(1)
  + Search = O(k) + O(n/k)
* Priority queue (heap)
  + Enter = O(log n)
  + Leave = O(log n)
* BST
  + Print = O(n)
  + Lookup/insert = O(log n)
  + Delete = O(log n)