*** CS 106X FINAL REFERENCE SHEET ***

Collections

Vector<T> (5.1)
`v.add(value);` or `v += value;`
`v.clear();`
`v.get(index)` or `v[index]`
`v.insert(index, value);`
`v.isEmpty()`
`v.remove(index);`
`v.set(index, value);` or `v[index] = value;`
`v.size()`
`v.toString()`

Stack<T> (5.2)
`s.clear();`
`s.isEmpty()`
`s.peek()`
`s.pop()`
`s.push(value);`
`s.size()`
`s.toString()`

Queue<T> (5.3)
`q.clear();`
`q.dequeue()`
`q.enqueue(value);`
`q.isEmpty()`
`q.peek()`
`q.size()`
`q.toString()`

PriorityQueue<T> (16.5)
`pq.changePriority(value, p);`
`pq.clear();`
`pq.dequeue()`
`pq.enqueue(value, priority);`
`pq.isEmpty()`
`pq.peek()`
`pq.peekPriority()`
`pq.size()`
`pq.toString()`

Set<T>, HashSet<T> (5.5)
`s.add(value);` or `s += value;`
`s.clear();`
`s.contains(value)`
`s.first()   // first element`
`s.isEmpty()`
`s.isSubsetOf(s2)`
`s.remove(value);`
`s.size()`
`s.toString()`
`a == b,  a != b`
`a + b,  a += b;` (union)
`a * b,  a *= b;` (intersection)
`a - b,  a -= b;` (difference)

Map<K, V>, HashMap<K, V> (5.4)
`m.clear();`
`m.containsKey(key)`
`m.get(key)` or `m[key]`
`m.isEmpty()`
`m.keys()`
`m.put(key, value)` or `m[k] = v;`
`m.remove(key);`
`m.size()`
`m.toString()`
`m.values()`

Grid<T> (5.1)
`g.fill(value);`
`g.get(row, col)` or `g[row, col]`
`g.inBounds(row, col)`
`g.numCols()   // or g.width()`
`g.numRows()   // or g.height()`
`g.resize(nCols, nRows);`
`g.set(row, col, value);` or `g[row][col] = value;`
`g.toString()`

Array List, Linked List, Binary Tree

```
class ArrayList {
public:
  void add(int value);
  void clear();
  int get(int index) const;
  void insert(int i, int v);
  bool isEmpty() const;
  void remove(int i, int v);
  void set(int i, int v);
  int size() const;
private:
    int* elements;
    int capacity;
    int mysize;
};
```

```
class LinkedList {
public:
    ...
private:
  ListNode* front;
};

struct ListNode {
  int data;
  ListNode* next;
  ListNode(int data = 0,
    ListNode* next = NULL);
};
```

```
class BinaryTree {
public:
    ...
private:
  TreeNode* root;
};

struct TreeNode {
  int data;
  TreeNode* left;
  TreeNode* right;
  TreeNode(int data = 0,
    TreeNode* left = NULL,
    TreeNode* right = NULL);
};
```

String Members and Utility Functions (3.2)

`s.at(index)` or `s[index]`	`s.length()` or `s.size()`	`equalsIgnoreCase(s1, s2)`
`s.append(str);`	`s.replace(index, len, str);`	`toLowerCase(str)`
`s.c_str()`	`s.rfind(str)`	`toUpperCase(str)`
`s.compare(str)`	`s.substr(start, length)` or `s.substr(start)`	`trim(str)`

`s.erase(index, length);`	`endsWith(str, suffix)`
	`startsWith(str, prefix)`
		`s1 + s2,  s1 += s2;`
		`s + c,    s += c;`

`s.find(str)`	`integerToString(int)`,
	`stringToInteger(str)`
		`s1 == s2,  s1 != s2,  s1 < s2,`
		`s1 <= s2,  s1 >= s2,  s1 > s2`

`s.insert(index, str);`	`realToString(double)`, `stringToReal(str)`	

*** CS 106X FINAL REFERENCE SHEET ***

Graphs

BasicGraph
g.addEdge(***v1, v2***); *g*.addEdge(***arc***); // or addArc
g.addVertex(***name***); // or addNode
g.clear();
g.containsEdge(***v1, v2***)
g.containsVertex(***name***)
g.getEdge(***v1, v2***) // NULL if none exists
g.getEdgeSet() // or getArcSet
g.getNeighbors(***v***) // set of Vertex*
g.getVertex(***name***) // or getNode
g.getVertexSet() // or getNodeSet
g.isEmpty()
g.isNeighbor(***v1, v2***) // direct neighbors

g.removeEdge(***v1, v2***); // or removeArc
g.removeEdge(***e***);

g.removeVertex(***v***); // or removeNode
g.resetData();
g.size() // number of vertices
g.toString()

Vertex (Node)
string name
Set<Edge*> edges
double cost (or weight)
bool visited
Vertex* previous
Color getColor()
setColor(***color***)
void resetData()
string toString()

Edge (Arc)
Vertex* start
Vertex* finish (or end)
double cost (or weight)
bool visited
string toString()

```
function dfs(v1, v2):
  mark v1 as visited.
    perform a dfs from each of v1's
       unvisited neighbors n to v2:
    if dfs(n, v2) succeeds: a path is found!
```

```
function bfs(v1, v2):
  create a queue of vertexes to visit,
    initially storing just v1.
  mark v1 as visited.
  while queue is not empty and v2 is not seen:
    dequeue a vertex v from it,
    mark that vertex v as visited,
    and add each unvisited neighbor n of v to queue.
```

```
function dijkstra(v1, v2):
   consider every vertex to have a cost of infinity,
   except v1 which has a cost of 0.
   create a priority queue of vertexes, ordered
   by heuristic, storing only v1

  while the pqueue is not empty:
    dequeue vertex v from pqueue, mark as visited.
    for each of the unvisited neighbors n of v,
     we now know that we can reach this neighbor
     with a total cost of (v's cost + the weight
     of the edge from v to n).
        if the neighbor is not in the pqueue, or
        this is cheaper than n's current cost, we
        should enqueue the neighbor n to the
        pqueue with this new cost, and with v as
        its previous vertex.
when done, we can reconstruct the path from v2 back
to v1 by following the previous pointers.
```

```
function astar(v1, v2):
  consider every vertex to have a cost of infinity,
  except v1 which has a cost of 0.
  create a priority queue of vertexes, ordered
  by heuristic, storing only v1 with a priority
  of H(v1, v2).
  while the pqueue is not empty:
    dequeue vertex v from pqueue, mark as visited.
    for each of the unvisited neighbors n of v,
    we now know that we can reach this neighbor
    with a total cost of (v's cost + the weight
    of the edge from v to n).
        if the neighbor is not in the pqueue, or
        this is cheaper than n's current cost, we
        should enqueue the neighbor n to the
        pqueue with this new cost plus H(n, v2),
        and with v as its previous vertex.
when done, we can reconstruct the path from v2 back to v1
by following the previous pointers.
```

```
function kruskal(graph):
   remove all edges from the graph.
   make a PQ of all edges, based on their weight (cost).
   while the PQ is not empty,
      dequeue an edge from PQ.
      if that edge's endpoints aren't already connected to one another,
         add that edge into the graph.
```