Study Guide 2 - COMP15

Heaps

- good performance for special application of collection of objects organized by imporatnce or priority
- is a complete binary tree, so most of them are nearly always implemented using the array representation (every level in a complete binary tree except maybe the last is completely filled, and all nodes are as far left as possible)
- values stored in a heap are partially ordered
- **max-heap**: every node stores a value that is greater than or equal to the value of either of its children (the root stores the maximum value)
- **min-heap**: every node stores a value that is less than or equal to that of its children (the root stores the minimum value of the entire tree)
- useful data structure when you need to remove an object with lowest or highest priority
- a heap is written in an array because a binary tree can be stored in an array, but because a binary heap is always a complete binary tree, it can be stored compactly
- no space required for pointers and parent and children node can be found using arithmetic
- given their positions, we can determine the relative order for values of two nodes in the heap only if one is a descendant of the other
- logical view of the heap is a tree structure, while the physical implementation uses an array
- a heap containing **n** nodes will have height **O(log n)**
- the height of a heap with **n** nodes is the ceiling function of log(n+1)
- level order traversal of a heap is represented in an array, and you start at index 1 for an array
- **left child: 2i + 1**
- **right child: 2i + 2**
- **parent: i-1/2**
- **heapify_up: O(log n) - upper bound**
- **heapify_up: O(1) - lower bound**
- **heapify_down: O(log n) - upper bound**
- **heapify_down: O(1) - lower bound**
- **insert: O(log n)**, lower bound is constant
- **extract: O(log n)**, lower bound is constant
- inserting an item requires heapify up, which starts at the bottom and goes to the top
- extracting the root requires heapify down, which starts at the top and goes to the bottom

```
//Min Heap Heapify Up
//swap if needed
//recursively repeat
template<class E>
void Minheap<E>::heapify_up(int index)
{
    if (index == 0 || index > Heap<E>::length || Heap<E>::length == 0)
        return;
```

```
    E item = Heap<E>::heap[index];
    int parent_index = Heap<E>::get_parent_index(index);

    if (Heap<E>::heap[parent_index] > item)
    {
        E temp = Heap<E>::heap[parent_index];
        Heap<E>::heap[parent_index] = item;
        Heap<E>::heap[index] = temp;
        heapify_up(parent_index);
    }
}

//Min Heap Heapify Down
//Compare myself to left/right children
//Recursively repeat until I'm in place
template<class E>
void MinHeap<E>::heapify_down(int index)
{
    int left, right;
    Heap<E>::get_children_indices(index, left, right);

    int small_index = index;
    if (left < Heap<E>::length &&
        Heap<E>::heap[small_index] > Heap<E>::heap[left])
        small_index = left;
    if (right < Heap<E>::length &&
        Heap<E>::heap[small_index] > Heap<E>::heap[right])
        small_index = right;

    //swap if you need to
    if (small_index != index)
    {
        E temp = Heap<E>::heap[index];
        Heap<E>::heap[index] = Heap<E>::heap[small_index];
        Heap<E>::heap[small_index] = temp;
        heapify_down(small_index);
    }
}

//Max Heap Heapify Up
//swap if needed
//recursively repeat
template<class E>
void Maxheap<E>::heapify_up(int index)
{
    if (index == 0 || index > Heap<E>::length || Heap<E>::length == 0)
        return;

    E item = Heap<E>::heap[index];
    int parent_index = Heap<E>::get_parent_index(index);

    if (Heap<E>::heap[parent_index] < item)
    {
        E temp = Heap<E>::heap[parent_index];
        Heap<E>::heap[parent_index] = item;
        Heap<E>::heap[index] = temp;
        heapify_up(parent_index);
    }
}
```

```
//Max Heap Heapify Down
//Compare myself to left/right children
//Recursively repeat until I'm in place
template<class E>
void MaxHeap<E>::heapify_down(int index)
{
    int left, right;
    Heap<E>::get_children_indices(index, left, right);

    int small_index = index;
    if (left < Heap<E>::length &&
        Heap<E>::heap[small_index] < Heap<E>::heap[left])
        small_index = left;
    if (right < Heap<E>::length &&
        Heap<E>::heap[small_index] < Heap<E>::heap[right])
        small_index = right;

    //swap if you need to
    if (small_index != index)
    {
        E temp = Heap<E>::heap[index];
        Heap<E>::heap[index] = Heap<E>::heap[small_index];
        Heap<E>::heap[small_index] = temp;
        heapify_down(small_index);
    }
}
```

- en.wikipedia.org/wiki/Binary_heap

Graphs

- data structure that expresses relationships between objects
- objects are vertices and relationships are edges. G = (V,E) is a set V of vertices and a set E of edges
- An edge {u,v} indicates that vertices u and v are connected by an edge, and edges of a graph can have direction or weight, or they may not
- A tree is a type of graph. It has a root, directional edges, and no cycles. Otherwise, it is still a set of vertices related by a set of edges
- A graph is essentially a tree, but there is no root, (can choose a vertex to begin a procedure)
- A graph has no leaves (but some vertices have no outgoing edges)
- there may or may not be a directional relationship between vertices (parent/child)
- two vertices u and v are adjacent if there is an edge having u and v as its endpoints
- undirected graph: two vertices are neighbors if they are adjacent ; if two vertices u and v are joined by an edge, then u is adjacent to b and v is adjacent to u
- in a directed graph, vertex v is adjacent to u if they are joined by a single edge
- two ways to represent a graph:
  - **adjacency matrix:** a 2d array in which an entry in the array i,j has the weight of the edge if there is an edge from vertex i to vertex j, and it has 0 otherwise. An unweighted graph would have a 1 to indicate the oresence of an edge i, j.
  - **adjacency list:** a linked list for each vertex, which contains all its neighbors
- **topological sort:** linear ordering of its vertices such that for every directed edge

u,v from vertex u to vertex v, u comes before v in the ordering. It is a partial-ordering of the vertices of a graph ensuring that dependencies are captured

- **topo-sort** applies to directed graphs w/o cycles (Directed Acyclic Graphs - DAGs)
  - an ordering of vertices in a directed acyclic grph, such that if there is a a path from vi to vj, then vj appears after vi in the ordering
  - a topological ordering is not possible if the graph has a cycle
  - simple algorithm for finding topological sort: find any vertex with no incoming edges. print this vertex and remove it, along with its edges, from the graph. Then apply this strategy to the rest of the graph
- a **path** is a sequence of vertices, and the length is the number of edges on that path; a path length of 0 is created when a vertice is directed towards itself
- a **simple path** is a path such that all vertices are distinct
- an **acyclic** graph has no cycles
- an **undirected** graph is connected if there is a path from every vertex to every other vertex. A directed graph with this property is **strongly connected**
- if a directed graph is not strongly connected but the underlying graph is connected, then the graph is **weakly connected**
- a **complete** graph is a graph in which there is an edge between every pair of vertices

Space Complexity of Graph

- space requirement of an adjacency matrix is O(V^2)
- adjacency matrix is always symmetric for an undirected graph
- if the graph is not dense, an adjacency list of space requirement O(E+V) is required

Time Complexity of Graph

- **Adjacency List:**
  - **find_vertex:** O(V)
  - **find_edge:** O(E) if know which linked list to look in
  - **find_all_neighbors:**
  - **report_path:** O(V+E)
- **Adjacency Matrix:**
  - **find_vertex:** O(V)
  - **find_edge:** O(V) if one vertex is found and need to check for adjacency, O(1) if both vertices are found and poistions are known in matrix
  - **find_all_neighbors:**
  - **report_path:** O(V^2)

BFS

- starting vertex s
- destination vertex d
- figuring out how to get to a specific vertex within a graph
- BFS guarantees the shortest path from s to d (one of the many shortest paths)
- is called breadth-first because it finds every vertex reachable from s in k steps before it finds anything reachable in k+1 steps

- Graph class stores vertices in a 1-dimensional array and edges in a 2D array
- start with all vertices unmarked. label a vertice marked once it has been explored
- **Auxiliary Data Structures:**

  - **Queue One:** Primary queue. Source s is first thing to be enqueued. Dequeue a vertex from Queue one when you are ready to explore its neighbors
  - **Queue Two:** "Neighbor Queue" - each time Queue One dequeues a vertex to be explored, store all its neighbors on Queue Two. Dequeue all of the items off Queue Two to complete exploring the current vertex from Queue One
  - **Predecessor Array:** Helps track overall path from s to d. At any moment, there exists a "current vertex" in Queue one, whose neighbors are explored in Queue Two. For eaxh of the neighbors, its predecessor in the path from s to d is the current vertex. Predecessor array is constructed as so: pos. 0 holds A's predecessor, 1 holds B's predecessor, 2 holds C's predecessor and so on. -1 is used to indicate a vertex has no predecessor
- algorithm:

  - find all of s's neighbors reachable in one step
  - if d is one of them, we are done
  - find all of s's neighbors (if we haven't found them already) that are reachable in two steps
  - if d is one of them we are done
- search is directed or undirected and always unweighted

```
void Graph::BFS(int s) {
        Queue Q;

        /** Keeps track of explored vertices */
        bool *explored = new bool[n+1];

        /** Initailized all vertices as unexplored */
        for (int i = 1; i <= n; ++i)
            explored[i] = false;

        /** Push initial vertex to the queue */
        Q.enqueue(s);
        explored[s] = true; /** mark it as explored */
        cout << "Breadth first Search starting from vertex ";
        cout << s << " : " << endl;

        /** Unless the queue is empty */
        while (!Q.isEmpty()) {
        /** Pop the vertex from the queue */
            int v = Q.dequeue();

            /** display the explored vertices */
            cout << v << " ";

        /** From the explored vertex v try to explore all               the
        connected vertices */
            for (int w = 1; w <= n; ++w)

                /** Explores the vertex w if it is connected to                     v
                and and if it is unexplored */
                    if (isConnected(v, w) && !explored[w]) {
                        /** adds the vertex w to the queue                          */
```

```
                        Q.enqueue(w);
                        /** marks the vertex w as visited */
                        explored[w] = true;
                    }
            }
        cout << endl;
        delete [] explored;
}
```

Abstract Classes and Templates

- an abstract class is specifically designed to be a base class and nothing more
- no instances of an abstract class can be created
- no such thing as a plain heap as a heap has to know whether it cares about minimum or maximum values, so the Heap class is defined as abstract
- an abstract class is distinguished by having at least one purely virtual function (expected to be overridden in derived classes, and does not get defined in the base)
- declare a virtual function in an abstract class like so:

```
class Heap
{
    public:
        virtual void heapify_down(int) =0;
};
```

  - the =0 at the en indicates there is no definition of the function in the base class
  - and that it is an abstract class
  - can use generic class template to substitute for any actual type without repeating code
  - example in Heap.h:

```
template <class E> class Heap
{
    public:
        void heapify_up(int, E) = 0;
        void insert_item(E);
}

//example in Heap.cpp
template class Heap<int>;
template class Heap<string>;
template <class E> void Heap<E>::insert_item(E item)
    {
        heap[length] = item;
        length++;
    }
```