CSCI 104L Lecture 8: Templates and Graphs

Templates

We've implemented a Linked List that stores ints: how could we extend it to allow doubles, chars, or strings?

```
template <class T>
struct Item {
  T value;
  Item<T> *prev, *next;
};
template <class T>
class LinkedList {
  public:
    LinkedList();
    LinkedList(T n);
    virtual ~LinkedList();
    void remove (Item<T> *toRemove);
  private:
    Item<T> *head;
};
template <class T>
LinkedList<T>::prepend (T item) { ... }
int main() {
  Item<string> *it = new Item<string>;
  LinkedList<string> *ll = new LinkedList<string>;
}
```

You can have multiple template types.

```
template <class keyType, class valueType>
class map { ... };
```

The main problem that arises is linker errors when you construct templated classes in the normal way. To get around this, you should put your entire implementation in the .h file. This is bad programming practice, so you should never do this for non-templated classes.

Graphs

A graph consists of a set of vertices/nodes V, and their edges E. By convention, $|V| = n$ and $|E| = m$

Question 1. Are these problems naturally modeled as directed or undirected graphs?

(a) Computer networks

(b) The Internet.

(c) Social networks.

(d) Road systems.

(e) Predator behavior between species

For a **Graph ADT**, we want to be able to do (at minimum) the following:

1. Add a node.
2. Delete a node.
3. Add an edge.
4. Delete an edge.
5. Test if an edge from u to v exists.
6. Enumerate all outgoing edges from a node.
7. Enumerate all incoming edges from a node.

Question 2. What are the runtimes of adding/deleting an edge, testing an edge, or enumerating edges, if we store the edges as....

- An unsorted array or linked list?
- A sorted array?
- An **Adjacency list**: for each node, store a list of adjacent nodes.
- An **Adjacency matrix**: in an n by n matrix of bools, A[u,v] indicates whether there is an edge from node u to node v.

In **sparse** graphs ($m = O(n)$), an adjacency list is more economical. For **dense** graphs ($m = \Omega(n^2)$), you might as well go for the adjacency matrix.

Breadth First Search

```
//nodes are named 0 through n-1
int d[n]; //stores distances from u
int p[n]; //stores paths
void BFS(int u) {//u is the start node
  enqueue node u
  d[u] = 0;
  while the queue is not empty {
    dequeue the next node v
    for all outgoing edges (v,w) from v {
      if we haven't yet visited w {
        d[w] = d[v]+1;
        p[w] = v; //tells us which node led to w
        enqueue node w
      }
    }
  }
}
```

Breadth-First Search is a simple algorithm that finds the shortest path (in terms of number of edges) from a given node u to all other nodes. BFS explores the graph in layers.

- Layer 0 consists only of the node u.
- Layer 1 consists of all nodes v with a direct edge $(u, v) \in E$.
- Layer k consists of all nodes w with a direct edge $(v, w) \in E$ from some node v in layer k-1. It does not revisit nodes from previous layers.
- What ADT/Data Structure will be needed to implement this?

Question 3. : What is the runtime of BFS?

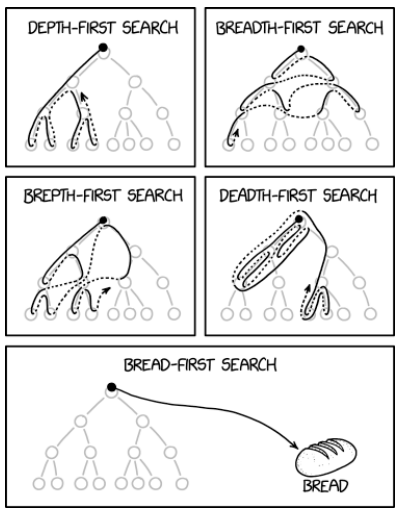
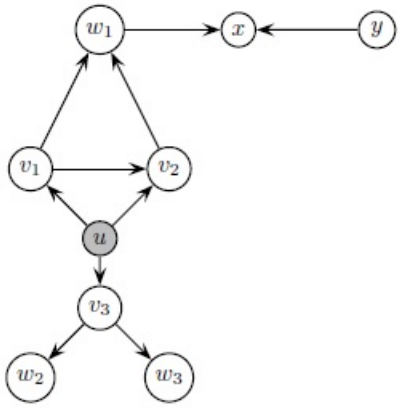


Figure 1: XKCD # 2407. A death-first search is when you lose your keys and travel to the depths of hell to find them, and then if they're not there you start checking your coat pockets.

Depth First Search

```
//nodes are named 0 through n-1
int p[n]; //stores paths
void DFS(int u) {//u is the start node
  mark u as visited
  for all outgoing edges (u,v) from u {
    if v has not been visited {
      p[v] = u;
      DFS(v);
    }
  }
}
```



Question 4. How would DFS explore the graph?

Question 5. What ADT/Data Structure will be needed to implement this?

Question 6. Does this find the shortest path?

Question 7. What is the runtime of DFS?