# CSE 101
# Winter 2023
# Midterm Exam 1 Solutions

1. (20 Points) Using only the List ADT operations defined in the project description for pa1, write a *client* function with the heading

```
List Shuffle(List A, List B)
```

Your function will return a newly allocated List containing the elements of Lists $A$ and $B$, shuffled together, one from $A$, then one from $B$, alternating between the two. If one of the Lists is exhausted before the other, then continue adding from the longer List, so that the returned List contains all elements from both $A$ and $B$. For instance, if $A = [1, 2, 3]$ and $B = [4, 5, 6, 7, 8]$ then Shuffle() will return the List $[1, 4, 2, 5, 3, 6, 7, 8]$. Shuffle() will make no changes to the states of its two arguments $A$ and $B$. Shuffle() has no preconditions.

**One of Several Possible Solutions:**

```
List Shuffle(List A, List B){
   List L = newList();

   moveFront(A);
   moveFront(B);
   while( index(A)>=0 && index(B)>=0 ){
      append(L, get(A));
      append(L, get(B));
      moveNext(A);
      moveNext(B);
   }
   while( index(A)>=0 ){
      append(L, get(A));
      moveNext(A);
   }
   while( index(B)>=0 ){
      Append(L, get(B));
      moveNext(B);
   }

   return L;
}
```

2. (20 Points) Using only the List ADT operations defined in the project description for pa1, write a *client* function with the heading

```
List Find(List L, int x)
```

Your function will return a new List consisting of all index positions within $L$ at which the element $x$ is located. For instance, if $L = [1, 5, 1, 6, 4, 1, 3, 1, 2]$ and $x = 1$, then Find() will return the List $[0, 2, 5, 7]$. If $x$ is not contained in the argument List $L$, then Find() will return an empty List. Find() will make no changes to the state of its argument $L$, and has no preconditions.

**One of Several Possible Solutions:**

```
List Find(List L, int x){
   List S = newList();

   moveFront(L);
   while( index(L)>=0 ){
      if( x==get(L) ){
         append(S, index(L));
      }
      moveNext(L);
   }

   return S;
}
```

3. (20 Points) Given a connected (undirected) graph $G$, and a vertex $x$ in $G$, the *eccentricity* of $x$ is the maximum possible distance from $x$, to any other vertex $y$ in $G$, i.e.

$$\text{eccentricity}(x) = \max\{\, \delta(x, y) \mid y \in V(G) \,\}$$

Using only the Graph ADT operations defined in the project description for pa2, write a *client* function with the heading

```
int Eccentricity(Graph G, int x)
```

Your function will compute and return the eccentricity of vertex $x$ within Graph $G$.

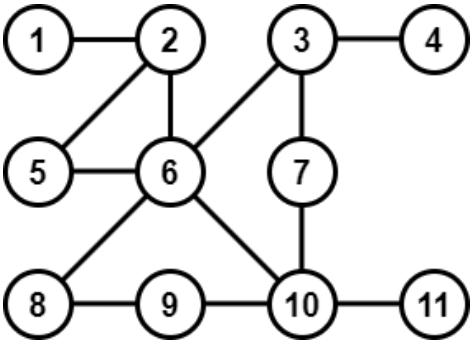
**One of Several Possible Solutions:**

```
int Eccentricity(Graph G, int x){

   int max, y;

   BFS(G, x);
   max = getDist(G, 1);
   for(y=2; y<=getOrder(G); y++){
      if( getDist(G, y)>max )
         max = getDist(G, y);
   }
   return max;
}
```

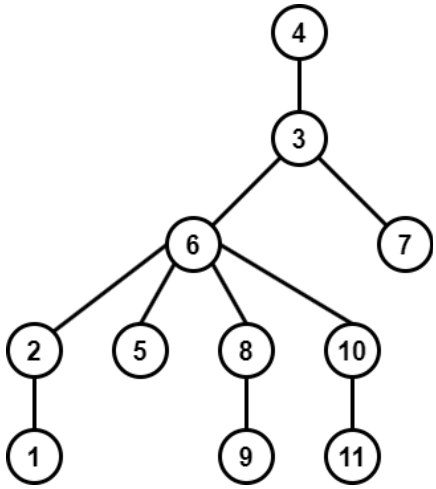
4. (20 Points) Run the BFS algorithm on the graph pictured below, with vertex **4** as the source. Fill in the table giving the adjacency list representation, colors, distances from the source, and parents in the BFS tree. List the discovered vertices in the order that they enter the queue. Draw the resulting BFS tree.



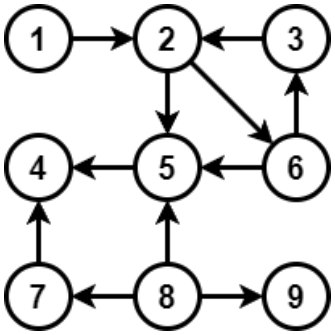

| *vertex* | *adj* | *color* | *distance* | *parent* |
|---|---|---|---|---|
| **1** | **2** | **black** | **4** | **2** |
| **2** | **1 5 6** | **black** | **3** | **6** |
| **3** | **4 6 7** | **black** | **1** | **4** |
| **4** | **3** | **black** | **0** | **nil** |
| **5** | **2 6** | **black** | **3** | **6** |
| **6** | **2 3 5 8 10** | **black** | **2** | **3** |
| **7** | **3 10** | **black** | **2** | **3** |
| **8** | **6 9** | **black** | **3** | **6** |
| **9** | **8 10** | **black** | **4** | **8** |
| **10** | **6 7 9 11** | **black** | **3** | **6** |
| **11** | **10** | **black** | **4** | **10** |

**Queue: 4 3 6 7 2 5 8 10 1 9 11**

**BFS Tree:**

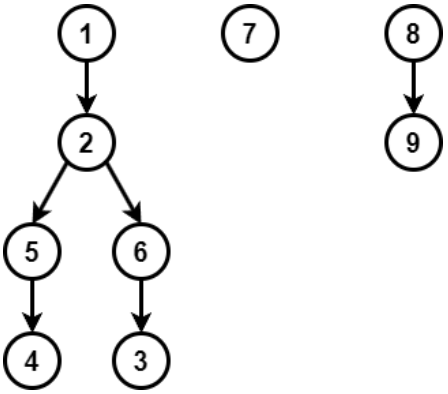


5. (20 Points) Run the DFS algorithm on the digraph pictured below. Process vertices in the main loop of DFS() by increasing vertex label. Process vertices in the for loop of Visit() by increasing vertex labels. As vertices finish, push them onto a stack. Fill in the table below giving the adjacency list representation, discover times, finish times and parents in the DFS forest. Draw the resulting DFS forest, and show the state of the stack when DFS is complete. Classify all edges as of type *tree*, *back* *forward* or *cross*. List all directed cycles that this digraph contains.




| *vertex* | *adj* | *discover* | *finish* | *parent* |
|---|---|---|---|---|
| **1** | **2** | **1** | **12** | **nil** |
| **2** | **5 6** | **2** | **11** | **1** |
| **3** | **2** | **8** | **9** | **6** |
| **4** | | **4** | **5** | **5** |
| **5** | **4** | **3** | **6** | **2** |
| **6** | **3 5** | **7** | **10** | **2** |
| **7** | **4** | **13** | **14** | **nil** |
| **8** | **5 7 9** | **15** | **18** | **nil** |
| **9** | | **16** | **17** | **8** |

**DFS Forest:**




**Stack:**

**8**
**9**
**7**
**1**
**2**
**6**
**3**
**5**
**4**

**Edge Classification:**

**Tree: (1,2) (2,5) (2,6) (5,4) (6,3) (8,9)**

**Back: (3,2)**

**Forward:**

**Cross: (6,5) (7,4) (8,7) (8,5)**

**Directed Cycles: (2 6 3 2)**