羅費南 – B06902102

Homework 4

Problem 1:

1.

a.

0	1	0	0
1	0	1	0
1	0	0	1
0	1	0	0

b.

1	1	0	1
1	2	1	0
2	0	2	0
1	1	0	1

c.

i	Paths from 1 to i
1	1->2, 2->3, 3->1
2	1->2, 2->1, 1->2
3	None
4	1->2, 2->3, 3->4

d. Source: https://www.um.edu.mt/library/oar/bitstream/handle/123456789/24439/powers%20of%20the%20adjacency%20matrix.pdf?sequence=1&isAllowed=y

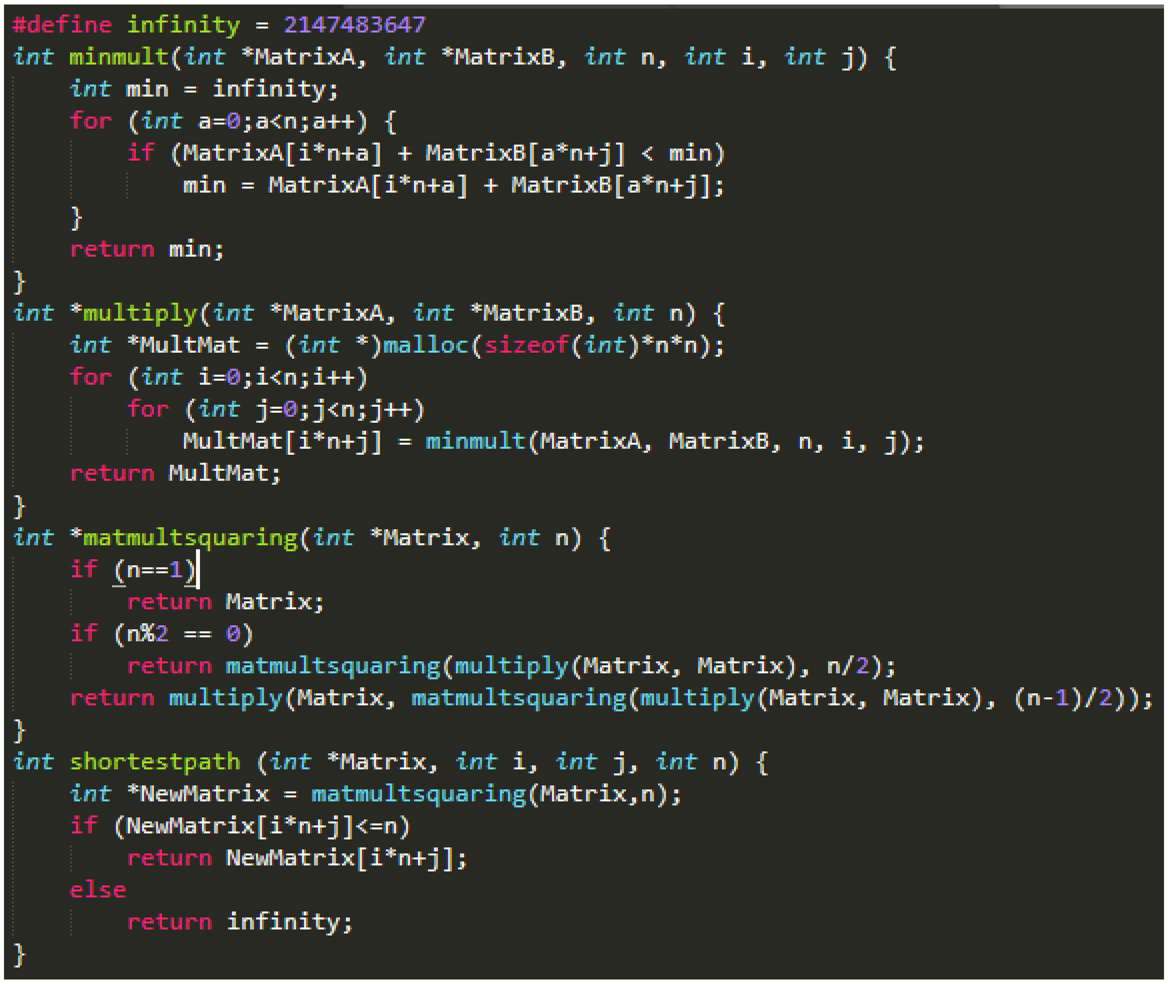
First we look at the case k=1. For k=1, $M^k = M$, and there will always be a walk of length 1 between i and j if and only if $M_{ij} = 1$, thus the result holds. Let's assume it also holds for k = n. We take the matrix $M^{n+1} = M^n * M$. By induction, we have that the i,j entry of M^n counts how many paths of length n exist between i and j. For the number of paths of length n+1 between i and j, we have that it equals to the paths of length n from i to the elements adjacent to j. This, we know is the entry i,j in $M^n * M$, which is equal to M^{n+1}. In M, the non-zero entries of M show which entries i,j are adjacent to each other, which is what we first proved in the case k=1. Thus, by induction the result is true for all positive values of n.

2.

a. First we look at the case k=1. For k=1, $M'^k = M'$, and since we defined the matrix to have 1 as a value for M'_{ij} if <i, j> exists, this will also be the length of the shortest path between i and j, then it holds for k=1. Let's assume it holds for k=n. First let's notice that $(M'^k)_{i, j}$ when i=j will always be 0, as when the multiplication is done, for k=i=j, the sum will be 0, which will always be the minimum value for an element in the matrix. Now for the case $M'^{n+1} = M'^n * M'$, we have that every non infinity entry in M'^n in the i, j position accounts for the shortest path between i and j by induction. For the shortest path between i and j in the M'^{n+1} matrix, it's the same as accounting for the shortest path from i to an adjacent node to j, which we'll call x, and the path existing between x and j of length 1 (given by the matrix M', for k=1, which we've already proven to be true). If such paths exist, they will have a non-infinity value, (1 for M', any value between 1 and k for

M'^n). During the multiplication operation, such values will be added up ($M'^n_{i,x} + M'_{x,j}$), and if they are the smallest value of the sum of all elements ($M'^n_{i,k} + M'_{k,j}$), they will denote the shortest path between i and j. Thus, by induction k=n holds for all positive values of n.

b.



```
#define infinity = 2147483647
int minmult(int *MatrixA, int *MatrixB, int n, int i, int j) {
    int min = infinity;
    for (int a=0;a<n;a++) {
        if (MatrixA[i*n+a] + MatrixB[a*n+j] < min)
            min = MatrixA[i*n+a] + MatrixB[a*n+j];
    }
    return min;
}
int *multiply(int *MatrixA, int *MatrixB, int n) {
    int *MultMat = (int *)malloc(sizeof(int)*n*n);
    for (int i=0;i<n;i++)
        for (int j=0;j<n;j++)
            MultMat[i*n+j] = minmult(MatrixA, MatrixB, n, i, j);
    return MultMat;
}
int *matmultsquaring(int *Matrix, int n) {
    if (n==1)
        return Matrix;
    if (n%2 == 0)
        return matmultsquaring(multiply(Matrix, Matrix), n/2);
    return multiply(Matrix, matmultsquaring(multiply(Matrix, Matrix), (n-1)/2));
}
int shortestpath (int *Matrix, int i, int j, int n) {
    int *NewMatrix = matmultsquaring(Matrix,n);
    if (NewMatrix[i*n+j]<=n)
        return NewMatrix[i*n+j];
    else
        return infinity;
}
```

The function minmult calculates the value to put in a cell as a result of the matrix multiplication defined in this exercise. The function multiply operates a multiplication of matrixes between MatrixA and MatrixB, but calls minmult to do the operation for each cell of the new matrix created. The function matmultsquaring does the matrix exponentiation by squaring algorithm, returning the original matrix if n=1; if n is even it returns the recursive call of the matrix resulting in squaring the original matrix and n/2 for the next call; and if n is odd it returns the multiplication of the original matrix, a call of the matmultsquaring function with the squaring of the original matrix and (n-1)/2 as the next power. Shortestpath calls the matmultsquaring function with n as the power, and reads the resulting value. If no path exists, then the result will be infinity. Otherwise, the shortest path will be stored in Matrix[i][j] (which is NewMatrix[i*n+j] for a 1-D array representation). The time complexity for the matrix exponentiation by squaring is log n, and the time complexity for multiplying two matrixes is T(n), thus the total time complexity is O(T(n)log(n)). Exponentiation by squaring is log n because we divide the problem in half for each recursive call, which is log(n).

Problem 2:

- Round 1: For the first round, since the rabbit knows the value of n, we can construct a pseudo adjacency matrix on the string, setting the string in position k as 0 if no edge between vertex **k/n** and **k mod n** exists, and set it as 1 if it exists. The rabbit will then reconstruct the original graph using these consensus, making an edge between vertex i and j if and only if S[i*n+j] == 1. This will make E' == E, and by definition also G' == G. Time complexity for rebuilding the entire graph is O(n^2), since the rabbit would read the entire string, which is of length n^2.

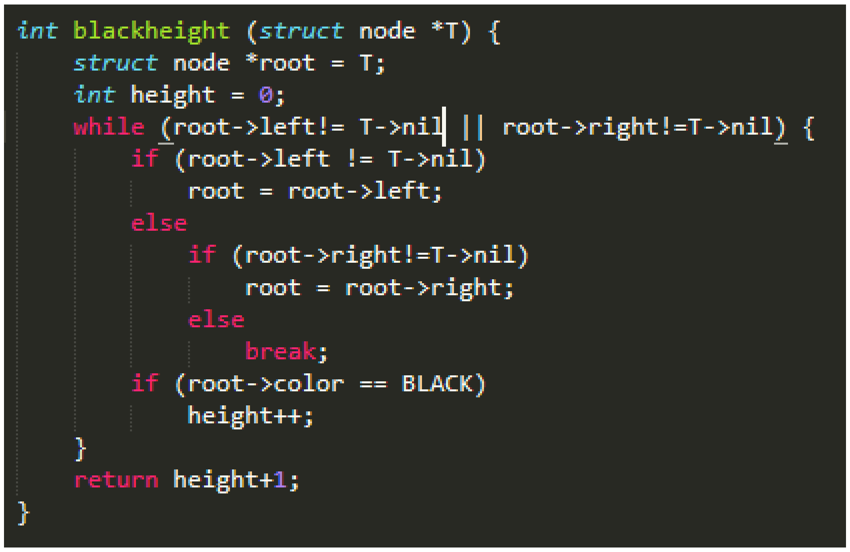
- Round 2: For the strategy of this round, I will refer to the use of Prufer code for tree creation. Since we can only pass strings as 0's and 1's, let's refer to each node number in binary. The length of this binary representation will be done in length ceil (lg n), since that's the position of the most significant bit in the binary representation of n. Now for the prufer code part; the process of encoding is to take the leaf of the tree with the least value and add the vertex to which this leaf is connected to the string in its binary representation. We proceed successively in this way until only two connected nodes are left; then we stop the process. Since we only save one value per node; and we save only n-2 nodes; the length of the string will be (n-2)* ceil (lg n).
  Now, let's describe the process for the rabbit to build the tree. The rabbit knows the value of n, so he can know the length of each number in binary representation. He creates an empty graph of size n, and knows that the string will have n-2 numbers in binary representation. For each number in the sequence we give to the rabbit, he will take the node with the lesser value not present in the Prufer code and that has not been yet added to the current tree and he makes an edge from the number in the Prufer code and the lesser node. He will repeat this step until he has read all values from the string. To save time, he can first translate all numbers into binary and then implement the above described algorithm. A representation using the

second link’s code is O(n log n) time for building the graph, and O(n log n) for translating all numbers, but since they are not done in the same loop we can say the entire operation can be done in O(n log n).
https://www.geeksforgeeks.org/prufer-code-tree-creation/
https://forthright48.blogspot.com/2018/01/prufer-code-linear-representation-of.html

- Round 3: For this round, since we know that a graph has at most |V|-1 edges, we need to define a way to use only the number of vertices available to denote all edges we can draw. Let’s use a method of writing this by 0’s and 1’s. One vertex will be represented by a pair of one 0 and a 1, written in that order. For each vertex in the tree, if the vertex has a child or multiple children, we will put its zero and one between the zero and one of the parent. Since the graph it’s a tree, we can always draw a path between any two vertices, so we can always make sure all paths can be drawn from any vertex we choose as long as the first vertex is a non-leaf vertex. Since we use two numbers per vertex, the length will be 2n. The time for building a tree this way is O(n), since it only needs to keep track of vertexes made on the run.

Problem 3:

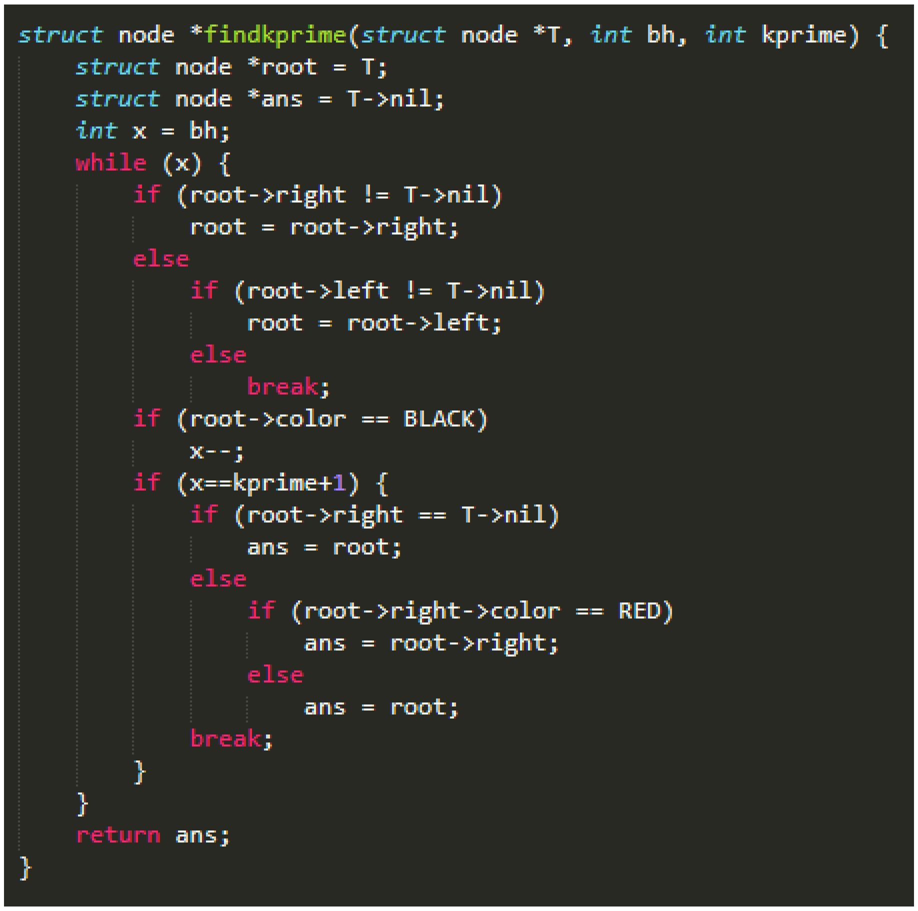
1. When building a RB-tree, we know that one of the properties is that its black height must be the same no matter what path down the tree we take, so if we find the black height amongst one path, we will find the black height of the entire tree. To do this, we can use the following algorithm.



```
int blackheight (struct node *T) {
    struct node *root = T;
    int height = 0;
    while (root->left!= T->nil || root->right!=T->nil) {
        if (root->left != T->nil)
            root = root->left;
        else
            if (root->right!=T->nil)
                root = root->right;
            else
                break;
        if (root->color == BLACK)
            height++;
    }
    return height+1;
}
```

It's O(h(T)) time because at any case we will traverse at most the entire height of the tree; since a RB-tree has a balanced black height by definition, the property will hold no matter what path we take.

2. Because of property 5 of red black trees, all paths will have the same black height. Since we already know the bh of the given tree T, if we want to find the key which value is equal to k', we will visit the right most node until its bh is equal to k'+1; beginning with the root, we will take 1 out of a counter value initialized to bh = k until k = k'+1. The right most one will have the biggest key amongst all of them with bh = k'.

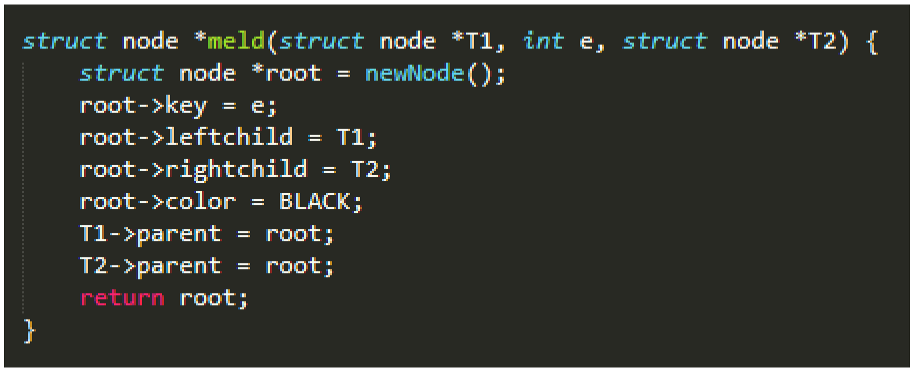


```c
struct node *findkprime(struct node *T, int bh, int kprime) {
    struct node *root = T;
    struct node *ans = T->nil;
    int x = bh;
    while (x) {
        if (root->right != T->nil)
            root = root->right;
        else
            if (root->left != T->nil)
                root = root->left;
            else
                break;
        if (root->color == BLACK)
            x--;
        if (x==kprime+1) {
            if (root->right == T->nil)
                ans = root;
            else
                if (root->right->color == RED)
                    ans = root->right;
                else
                    ans = root;
            break;
        }
    }
    return ans;
}
```

3. To prove this statement, we will look at some properties of red black trees. First, binary search trees will always have key elements smaller than it to the left and elements bigger than it to the right. Second, property 4 of red black trees state that if a node its red, both of its children will be black. Now, to find the node we will return in the previous function, we find the biggest node which bh is equal to k'+1. First we find the first node that has a bh equal to k'+1. This node might have two children, but we only care about its right child, since by the definition stated above the left child will have a smaller key and we want to find the biggest possible key. If the right child is a leaf, the case is trivial. If the right child is black instead, it is also trivial, since its black height will be of k' by the definition of black height. If the black node we find has a right red child, this child will also have a black height equal to k'+1 (since this node will not contribute to the parent's total black height). By property 4, this node will have two black children (either node or

leaf), and they will have a black height of k’ by definition of black height. This right red child will also be the biggest possible key, since if it’s a right child it will have a key bigger than its parent. This red right child will be the node we return in the function, and as proven above, its right child will have a bh of k’.

4.

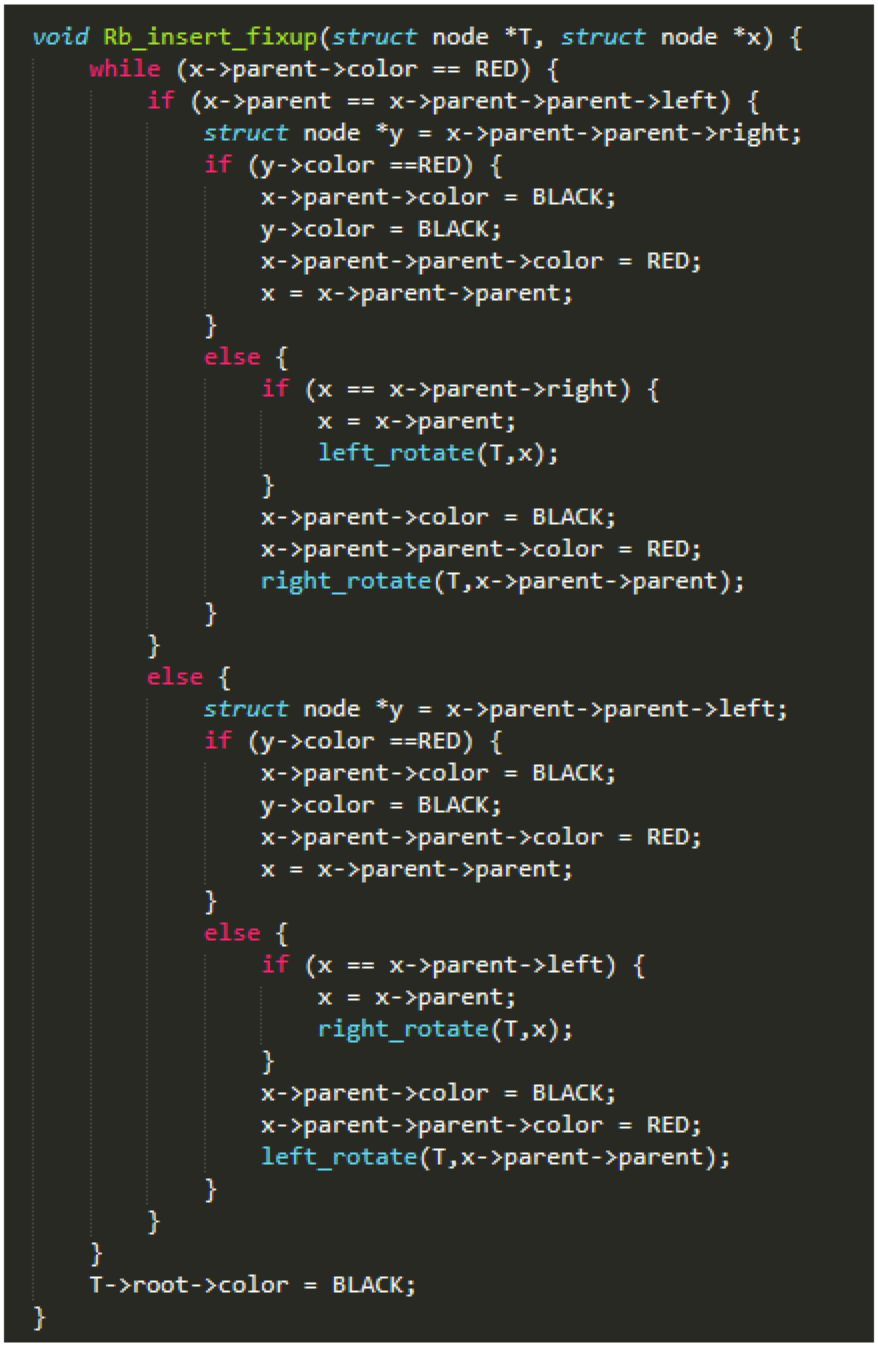


```c
struct node *meld(struct node *T1, int e, struct node *T2) {
    struct node *root = newNode();
    root->key = e;
    root->leftchild = T1;
    root->rightchild = T2;
    root->color = BLACK;
    T1->parent = root;
    T2->parent = root;
    return root;
}
```

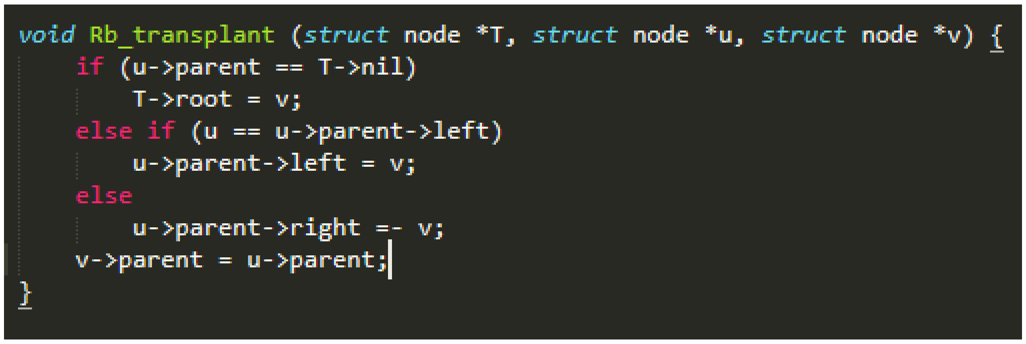
The resulting tree will also be a valid red black tree if T1 and T2 have the same bh, because setting up the root for this new tree as black also preserves the rules of red black trees; rule 1 and 2 are trivially preserved, since we color the root black, rule 3 is also preserved since we don’t modify anything related to the leaves, rule 4 is also preserved since that rule was preserved in T1 and T2 and we add a black node, rule 5 is also preserved because the bh for T1 and T2 is the same, and adding a node at the root doesn’t break the bh for any node in the tree. The resulting bh for the new root will be of the previous bh+1. The new root for the tree will contain e as its key, and point to T1 as left child (all values less than e) and T2 as right child (all values greater than e).

5. By performing meld(Ty, e, T2), we will have set a new tree, with a root T with the following values: T.key = e, T.left = Ty, Ty.parent = T, T.right = T2, T2.parent = T. To preserve rule 1, we need to set it to a color that is either red or black. To preserve rule 2, we don't need to make any change as long as the main root of the entire tree is not modified; since Ty is a subtree of the entire tree, we will not modify the root, and rule 2 will be preserved. Rule 3 is trivially preserved, since we won't modify the leaves. Rule 5 says that we need to preserve the entire black height of the tree, and the best way to do so is to set the value of the node containing e as red, since that won't modify the entire bh of the tree at any point (since the height of T2 and Ty is the same, k').

6. To preserve rule 4, we need to check for two different cases in the melding process and which node is the parent of Ty before the insertion. As we saw in question 2, the node with the biggest key and bh = k' can either be black or red, and its parent can also have red or black color if its black or black color if its red (It can't be red-red because of property 4). If the color of that node Ty is black and its parent is black too, by setting the root of T to red we already have a tree that doesn't violate rule 4 (since the root of T2 has to be black because of property 2). We can call a function called RB_Transplant (T, y, e) to fix any position issue on T. Now, for the coloring issue (because we colored the root as red, it might encounter red parents, so we need to fix that), we call RB_insert_fixup (T, y) to fix the coloring on the tree, which is what fixes the cases of wrong coloring in a normal red black tree insertion.



```c
void Rb_insert_fixup(struct node *T, struct node *x) {
    while (x->parent->color == RED) {
        if (x->parent == x->parent->parent->left) {
            struct node *y = x->parent->parent->right;
            if (y->color ==RED) {
                x->parent->color = BLACK;
                y->color = BLACK;
                x->parent->parent->color = RED;
                x = x->parent->parent;
            }
            else {
                if (x == x->parent->right) {
                    x = x->parent;
                    left_rotate(T,x);
                }
                x->parent->color = BLACK;
                x->parent->parent->color = RED;
                right_rotate(T,x->parent->parent);
            }
        }
        else {
            struct node *y = x->parent->parent->left;
            if (y->color ==RED) {
                x->parent->color = BLACK;
                y->color = BLACK;
                x->parent->parent->color = RED;
                x = x->parent->parent;
            }
            else {
                if (x == x->parent->left) {
                    x = x->parent;
                    right_rotate(T,x);
                }
                x->parent->color = BLACK;
                x->parent->parent->color = RED;
                left_rotate(T,x->parent->parent);
            }
        }
    }
    T->root->color = BLACK;
}
```

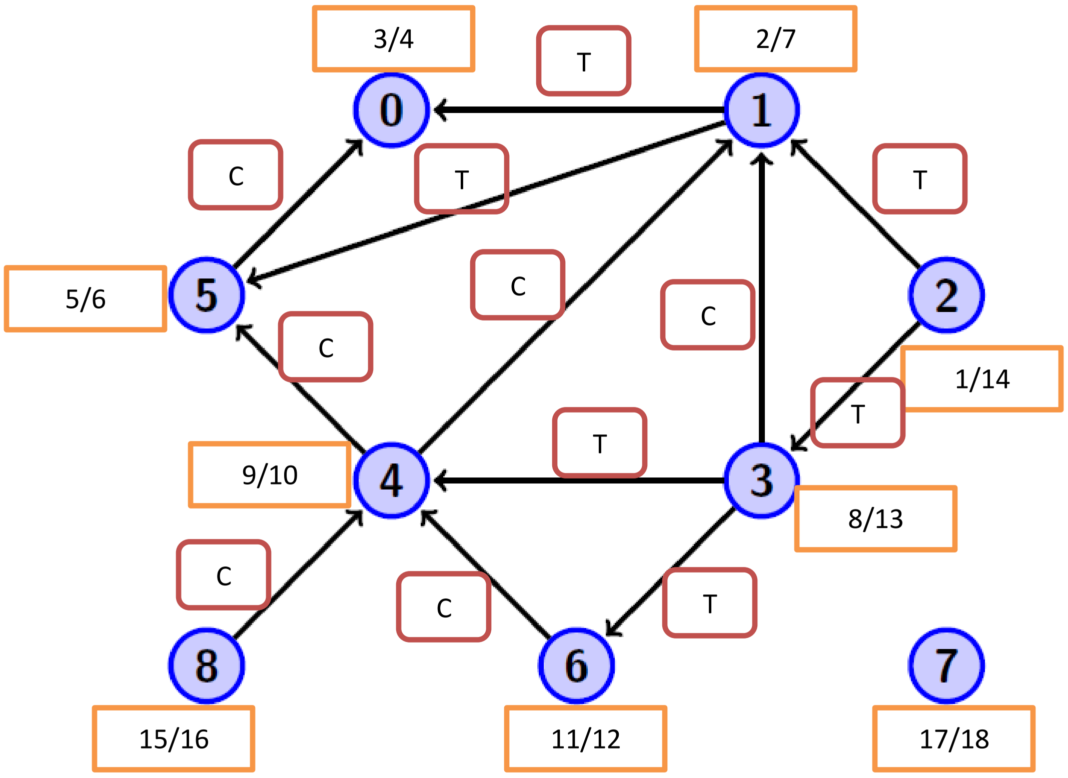


```c
void Rb_transplant (struct node *T, struct node *u, struct node *v) {
    if (u->parent == T->nil)
        T->root = v;
    else if (u == u->parent->left)
        u->parent->left = v;
    else
        u->parent->right =- v;
    v->parent = u->parent;
}
```

7. To implement the final meld algorithm, since we know that h1 is greater or equal than h2, we can first find the bh = k' value described above inside of T1, which takes O(h1) time. Then, we meld that sub-tree with T2, which takes O(1) time without any checking for correctness. For fixing up the tree as described in exercise 6, it takes O(h2) time. Since all processes run independently from each other, it will take O(h1 + h2) time, the correctness is proven by the problems above since we are doing the same processes.

Problem 4:

1.

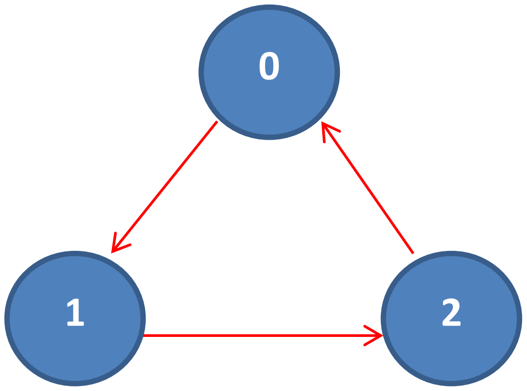


Sequence: {7, 8, 2, 3, 6, 4, 1, 5, 0}

2. Example of one sequence of u, such as cnt is minimal: {2, 8, 7, 0, 1, 3, 6, 4, 5}

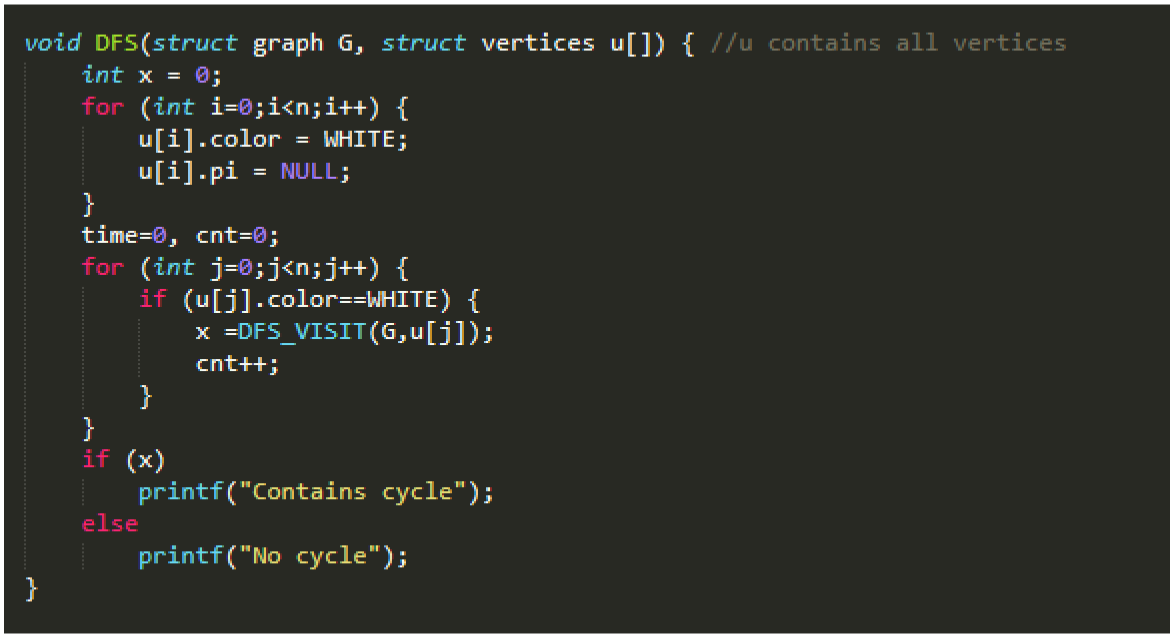
   This sequence was found by listing all vertexes with no incoming edges first (In order 0). This works because dfs-visit is called if the vertex you visit first is color white. The strategy here is to reduce the number of vertexes that are white after checking all the vertexes that can be accessed from the first vertexes checked in the sequence. Since no matter what our choice is, the vertexes with no incoming edges won't be able to be colored unless we specifically call them in the DFS visit sequence (line 10 in DFS), we will call them all first. After checking all of these vertexes, the remaining vertexes will already be black, since they have at least one edge coming from another vertex. Minimal value of cnt is 3.

3.

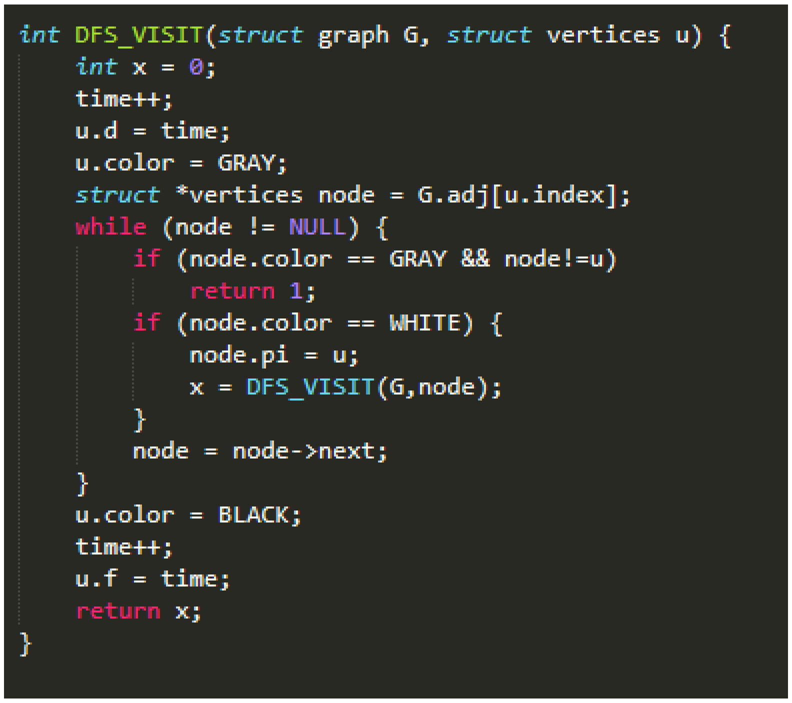


   This graph cannot be topologically sorted, since one of the requirements to perform a topological sort is for the graph to be a directed acyclic graph, and this graph contains a cycle. More technically, for every directed edge in the graph <u,v>, u has to come before v; and in the above graph, there's no way to guarantee this to happen (let's say we begin at vertex 0, then 2 will come after 0, but there's an edge from 2 to 0, so the above rule will not be possible).

4. Modify DFS-visit such that if it encounters a gray vertex that is adjacent to a vertex which we are currently visiting (v), and this vertex (u) is not the parent of the vertex we are now in, then we went through a path from vertex u to vertex v without using the existing edge <u, v>, thus the graph has a loop. If it happens to be this way, it will return it has a cycle, otherwise it will visit every vertex without finding the cycle. The running time is O(V), since if the number of edges E>V-1, then there must be a cycle in the graph and it can be detected in O(V) time (Since at edge number V it will definitely link to a vertex we have visited before).



```c
void DFS(struct graph G, struct vertices u[]) { //u contains all vertices
    int x = 0;
    for (int i=0;i<n;i++) {
        u[i].color = WHITE;
        u[i].pi = NULL;
    }
    time=0, cnt=0;
    for (int j=0;j<n;j++) {
        if (u[j].color==WHITE) {
            x =DFS_VISIT(G,u[j]);
            cnt++;
        }
    }
    if (x)
        printf("Contains cycle");
    else
        printf("No cycle");
}
```



```c
int DFS_VISIT(struct graph G, struct vertices u) {
    int x = 0;
    time++;
    u.d = time;
    u.color = GRAY;
    struct *vertices node = G.adj[u.index];
    while (node != NULL) {
        if (node.color == GRAY && node!=u)
            return 1;
        if (node.color == WHITE) {
            node.pi = u;
            x = DFS_VISIT(G,node);
        }
        node = node->next;
    }
    u.color = BLACK;
    time++;
    u.f = time;
    return x;
}
```

5. The statement is true. By definition of topological sort, for every edge in the graph <u,v>, we will have that u comes before v in the ordered list. Now, a preorder traversal of a tree is defined by visiting first the root of a tree, then its children; it will output first the nodes from which the edges begin (u). Since this is the same definition as a topological sort of the graph, they are equivalent.

6. The statement is true, since by definition of topological sort, a sequence which is topologically sorted will have that for each edge in the graph <u,v>, u will come before v in the output. Since the desired sequence to compare will have all of the parents appear before the children (such as in heaps), then this will hold for all T2 which are binary complete trees.

Problem 5

DSA was a course that taught me a lot about problem solving and fed my curiosity to learn new things and try to explore not so obvious solutions from the problems that were presented to us during the homework and tests. The class activities I think were great ways to make all of us use the skills we learned in class and turn them into some great thought exercises. Group activities were very effective at making us collaborate and share ideas to solve the many activities that were presented to us. Although homework was a little time consuming and sometimes hard, it was well designed to make us think about the problems that were presented to us, and not only replicate what they said in class in a poorly designed program. I think after taking this class my programming skills have greatly improved and can’t wait to tackle more challenges that come ahead of this class. I strongly believe this class will give me the necessary tools for the rest of my career and I think that Michael and his team of TA’s did a great job in leading us the right way. I also commend the responsiveness of the TA’s for helping me in particular resolving my doubts or helping in a constructive way to some confusion I might have had with the homework, which is always greatly appreciated.