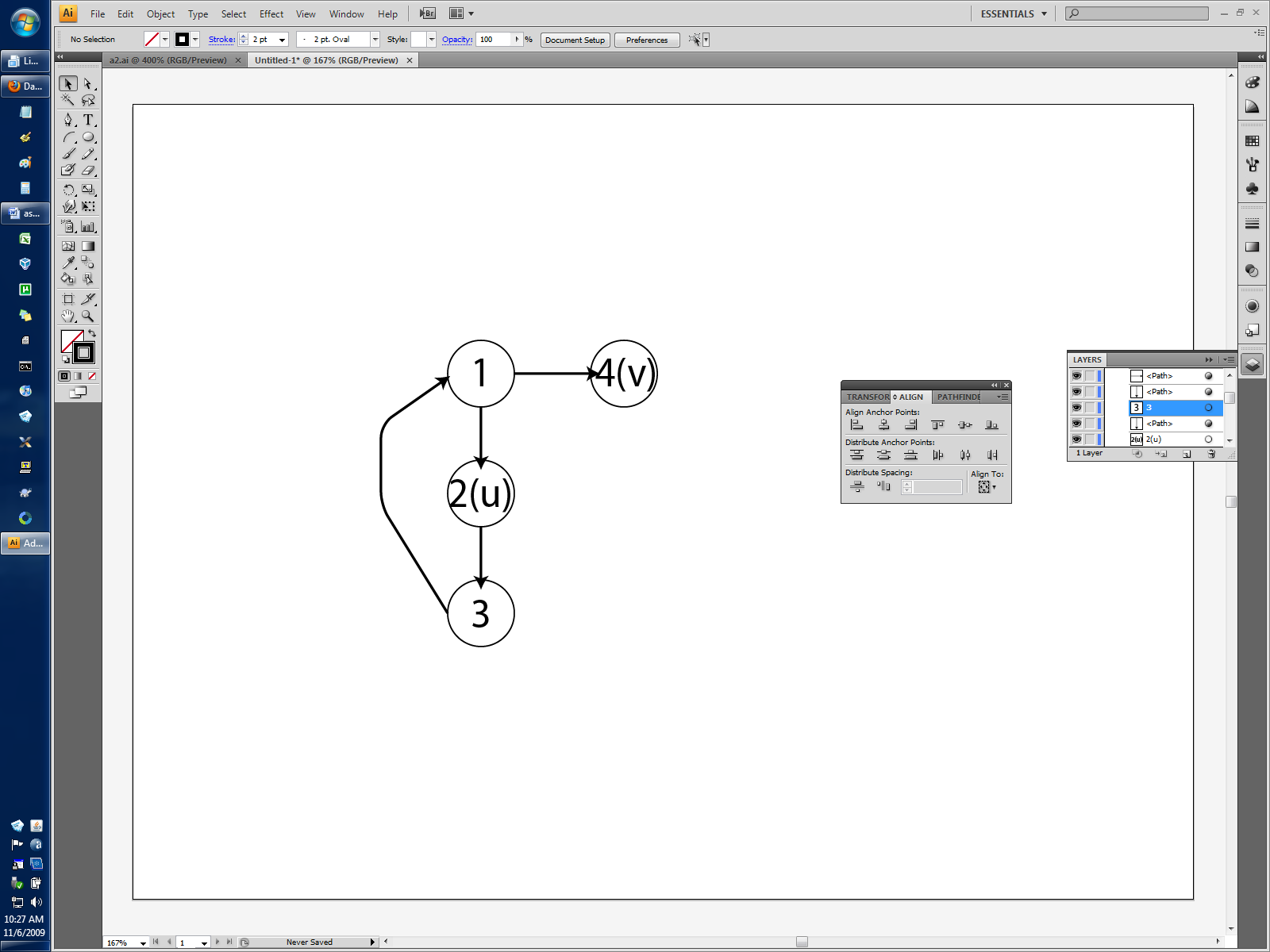
**Q1a)**

From the notes: [http://www.cs.mcgill.ca/~pnguyen/251F09/DFSproperty.pdf]

“*v* is a descendant of *u* in a DFS tree if and only if s[u] < s[v] < f[v] < f[u].”

If there is a path from u to v and s[u] < s[v]; then the only way to break the decendancy condition is if the visit of u finished before the search of v begins. This can easily be achieved in a cyclical graph

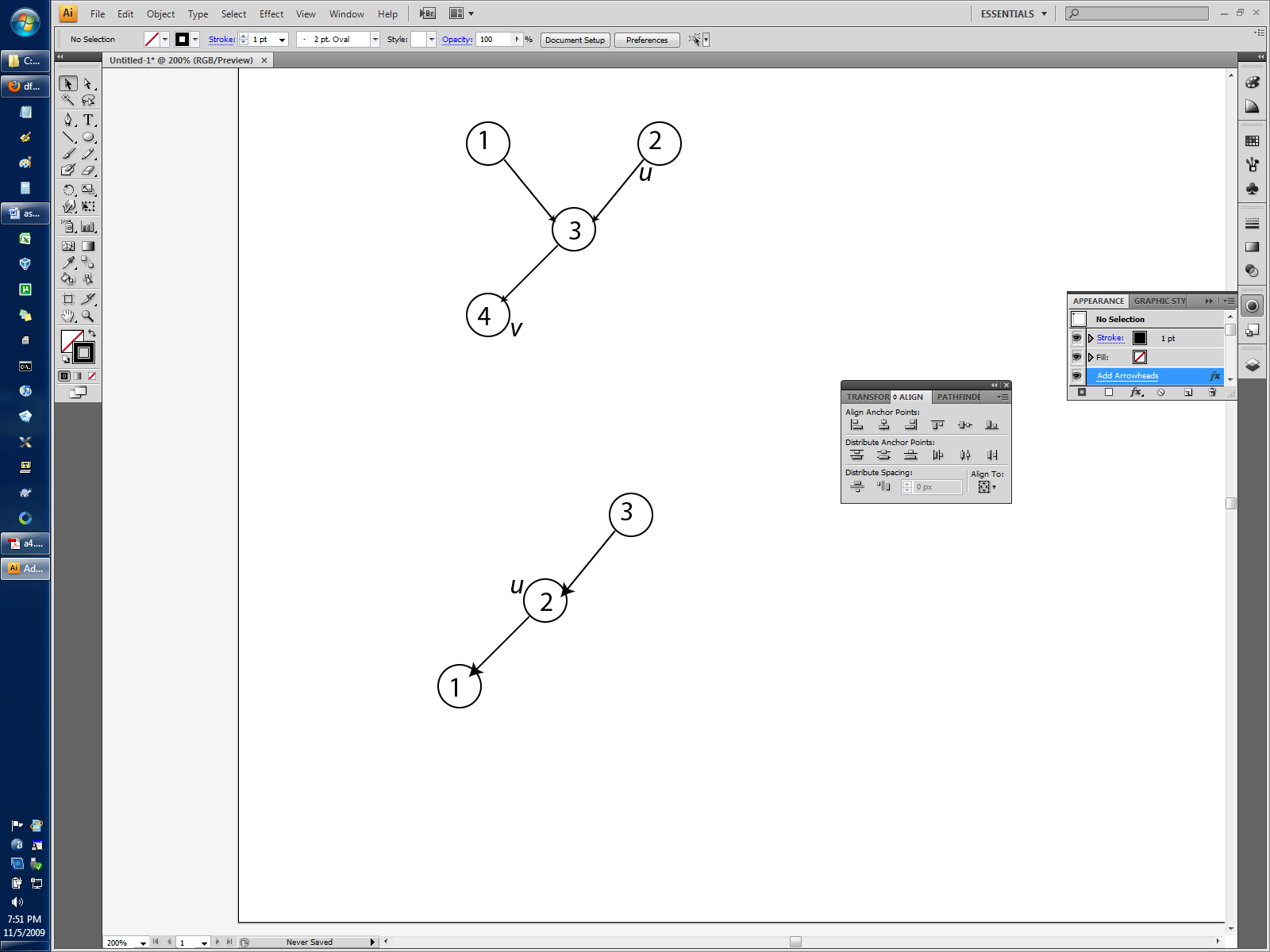


|  |  |
| --- | --- |
| Vertex | Adj |
| 1 | 2(u), 4(v) |
| 2 (u) | 3 |
| 3 | 1 |
| 4 (v) | NULL |

In this case, we can observe a path from u to v.

But f[u]<f[v] so v is NOT a descendant of u.

**Q1b)**



|  |  |
| --- | --- |
| Vertex | Adj |
| 1 | NULL |
| 2 (u) | 1 |
| 3 | 2 (u) |

In this case, u has an incoming edge from 3 and an outgoing edge to 1. The DFS will output 3 trees, each one containing a single node. So the tree of u will only contain u.

**Q2a)**

void TopoSort(int n, adj[n]){

int outdegree[n]; // Array to recors the out degree for every nodes from 1 to n

node incomming[n]; // Head of the linked list of incomming edges of node

L = new LinkedList(); // Empty list that will contain the sorted elements

reversedL = new srack(); // Empty stack that will contain sorted elements in reversed order

S = new LinkedList(); // List of all nodes with no incoming edges

// First we populate outdegree and incomming

for(int i = 1; i <= n; i++)

outdegree[n] = 0;

for(i = 1; i <= n; i++)

for all nodes u in adj[i] do{

incomming[u].add(i);

outdegree[i]++;

}

// Populate the list of all nodes with no incoming edges

for(i = 1; i <= n; i++)

if(outdegree[i] == 0)

S.add(i);

// Run the algorithm for all the nodes

while(!S.empty();){

node u = S.remove(); // First remode a node from the list of nodes with outdegree 0

reversedS.push(u); // Add it to the topo reversed sorted stack

// Since we 'removed' the node, update the outdegree of all incomming nodes

// If their outdegrees reaches 0; add them to S

for all node v in incomming[u]

if(--outdegree[v] == 0)

S.add(v)

}

// Look for a cycle; condition if any node still has an outdegree > 0

for(i = 1; i <= n; i++)

if(outdegree[i] > 0){

printf("Containing cycle at node: %d", i);

return;

}

// If there is no cycle found, reverse the order of the topological sort (since we sorted backwards)

while(reversedL.size() > 0)

L.add(reversedL.pop(););

// Finally print all the elements in topological sorted order

for every node v in L do

prinf("%d", v);

}

**Q2b1)**

Base case: If we have a graph with 2 nodes in a cycle; node 1 and node 2, where adj[1] = 2 and adj[2] = 1

Running the algorithm, both nodes will have outdegree = 1; so the list S will never contain any nodes. The main loop will be skipped, and we’ll get to the loop looking for a cycle: since there is at least one node with outdegree > 0 (in this case, both nodes will fulfill this condition), the algorithm will print an error.

Induction step: Assume the algorithm works for a graph of arbitrary size n. Every time we remove a node, we decrement the outdegree of all the nodes pointing to him, but we can only remove that node if its outdegree = 0. 2 nodes in a cycle will never have outdegree = 0, since they are in a cycle, so they will never be eliminated, so the algorithm will always detect them at the end.

**Q2b2)**

Base case: 2 nodes where adj[u] =v and adj[v] = null. Running the algorithm, outdegree[u] = 1 and outdegree[v] = 0, so node v will appear first on the stack, outdegree[u] will be decremented to 0 and node u will appear on the stack. The list is constructed by popping the stack, so the list will contain [u, v].

Induction step: Assuming the algorithm works for a graph of arbitrary size n, we always remove node with smallest outdegrees. If there is an edge from u to v, the nescessairly, when all the other nodes are removes, we are left with outdegree v > outdegree u, so they will appear in the right order in the list L.

**Q3)**

We’ll use a modification of the algorithm presented in the book and class notes:

1. Call DFS(G) to compute finishing times f[u] for each vertex u
2. Compute GT
3. Call DFS(GT), but in the main loop of DFS, consider the vertices in order of decreasing f[u] (as computed in line 1)
4. (modified) Finally compute B[i] using the array SCC constructed

SCC(n, adj[]){

Linked list L = NULL; // L will contain the vertices of G in decreasing order of f[v]

// Initialisation for the first DFS

for(int u = 1; u <= n; u++)

color[u] = White;

// 1. Call DFS(G) to compute finishing times f[u] for each vertex u

// Other the F[u] on the fly by adding them to the Linked List L as computed

// Main loop for first DFS

for(u = 1; u <= n; u++)

if(color[u] = white do

SCC-Visit1(n, ajd[n], u, L);

// 2. Compute G transpose by computing an array inc[]

// inc[n] contains the head pointer to all the incoming nodes to node n

for(u = 1; u <= n; u++)

inc[u] = NULL;

for(u = 1; u <= n; u++)

for all node v in adj[u]

inc[v].add(node u);

// 3. Call DFS(GT), but in the main loop of DFS, consider the vertices

// in order of decreasing f[u] (as computed in line 1 and stored in L)

// Initialisation for the second DFS

for(int u = 1; u <= n; u++)

SCC[v] = 0; // New array; SCC[v] is the SCC identifier for V

int k = 0; // # of SCCs

// Main loop for second DFS

for each node u in L do

if(color[u] == black){

k++;

SCC-Visit2(n, inc[], u, k, SCC[]);

}

// Finally, k is the number of SCCs, all we need is to compute B[i]

for(u = 1 to n)

for all node v in adj[u]

if(v is not in B[SCC[u]] && SCC[v] != SCC[u])

B[SCC[u]].add(v);

}

// The following function is copied from the class notes

CSS-Visit1(n, adj[], u, L){

stack S = null;

S.push(u);

while(!s.empty){

x = S.pop;

if(color[x] == White){

S[x] = ++time;

color[x] = gray;

S.push(x);

for each v in adj[x] do{

if(color[v] == White){

p[v] = x;

S.push(v);

}

else if(color[v] == Gray)

return false;

}

}

else if(color[x] == Gray){

f[x] = ++time;

color[x] = Black;

insert x into L;

}

}

}

// Following function is also pretty much copied from the notes, with the exception that we use the array inc[] instead of adj[], such that the search is done in GT

CSS-Visit2(n, inc[], u, k, SCC){

stack S = null;

S.push(u);

while(!s.empty){

x = S.pop;

SCC[x] = k;

if(color[x] == Black){

color[x] = Blue;

S.puch(x);

for each v in inc[x] do

if(color[v] == Black){

p[v] = x;

S.push(v);

}

}

else if(color[x] == Blue)

color[x] = Red;

}

}

**Q4a)**

**Extra Field:**

Add a field to every node that records the number of nodes in its left subtree. Let’s call it ‘lts’ (for left-sub-tree-size)

**Algorithm:**

Starting at the root node, we compare k and root.lts. There are 3 possibilities:

* If k <= root.lts, then recursively search for the kth smallest element in the left subtree.
* If k = root.lts + 1, then the root is the kth smallest element; return root node.
* If k > root.lts + 1, recursively search for the kth smallest element in the right subtree.

Pseudocode:

node findKth(int k, node root){

if ( k <= root.lts){

if(root.left == NULL)

return NULL;

else

findKth(k, root.left);

}

if ( k == root.lts + 1)

return root;

if (k > root.lts + 1){

if(root.right == NULL)

return NULL;

else

findKth(k, root.left)

}

}

**Running time:**

Since the algorithm starts at the root and is called recursively on a child node, it can only travel as far as a leaf node, which gives a running time equal to the height of the tree.

**Q4b)**

Essentially the same algorithm as a regular binary tree insert, except that we update the added left tree size (ltsz) field as the new node is trickled down to its new home in the tree. The following is a modified code originally taken from:

“http://www.roseindia.net/java/java-get-example/java-binary-tree-insert.shtml”

public void insert(Node T, node x) {

if (x.value < T.value) {

T.ltsz++;

if (T.left != null)

insert(T.left, x);

else {

System.out.println(" Inserted " + x.value + " to left of Node " + T.value);

T.left = x;

x.ltsz = 0;

}

}

else if (x.value > T.value) {

if (T.right != null)

insert(T.right, x);

else {

System.out.println(" Inserted " + x.value + " to right of Node " + T.value);

T.right = x;

}

}

}

**Q4c)**

The structure of this algorithm is the same as the algorithm presented in class, with one modification: we keep track and update the ‘its’ fields (left-sub-tree-size field)

BST\_Delete(T, x){

if( x.left == NULLL)

Transplant(T, x, x.right);

if( x.right == NULL){

// Update lts field because we are removing a node from the left sub tree

x.parent.lts--;

Transplant(T, x, x.left);

}

else{

y = BST\_sucessor(T, x);

if( y == x. right){

y.left = x.left;

// Swapp the lst fields

swapp(x.left.lst <=> y.left.lst);

Transplant(T, x, y);

}

else{

y.parent.left = y.right;

x.value = y.value; // Copy data, keep pointers

}

}

}

// Replaces node x by node y

Transplant(T, x, y){

// If x was root node, y is now root node; no update to lts

if(x.parent == null)

T.root = y;

// If x is left child and y is right child

else if(x == y.parent.left)

x.parent.left = y;

// If x is the right child and y is the left child

else

x.parent.right = y;

if(y != NULL)

y.parent = x.parent;

}

BST\_sucessor(T, x){

// Finds node with smallest key that is > x

// i.e. leftmost node in the right subtree

// This function is unchanged; see class notes for code

}