Problem 1.

(a) \( n = 10; k = 4 \). The colored nodes are the members of the specified group. The bold edges are the tree edges. A second tree can be formed by replacing the edges 1, 2 by 3, 4.

(b) In order to accomplish only one simple path between any two members of the group, we may connect them as a spanning tree. The spanning tree algorithm is to form a simple path from an initiator to every other nodes, so by using this algorithm guarantee that this is a simple tree.

The Algorithm:

```plaintext
Program Multicast
Define S[k] {the group of k nodes}
Initially parent = i,

{for the initiator process}
send probes to each neighbor;

{for a non-initiator process i}
do
if S[k] is empty -> Stop; fi // has form a simple path
probe ^ parent = i ->
    parent := sender;
    if i is a member of S[k], then delete i from the S[k]; fi
    if S[k] is empty -> Stop; fi
    if i is not a leaf -> send probes to non-parent neighbors; fi
od
```
This algorithm will terminate, since \( k \) is a finite. Every time one of the \( k \) nodes in \( S[k] \) will connect to the path, it will be deleted from the \( S[k] \) and when the \( S[k] \) becomes empty, the program will be terminate. This algorithm also eliminates unnecessary edges and nodes to the simple path. Hence there will be no cycle. As soon as the simple path includes every \( S[k] \) nodes, the program stops. So the simple path will only includes the necessary edge for connecting these \( k \) nodes.

**Problem 2.**

To elect \( k \) leaders, we can use a generalized version of Chang-Roberts Algorithm.

Processes are **red** or **blue**. Initially they’re all **red** i.e. they’re all candidates. Each process maintains an array of size \( k \), called \( \text{Leader}[k] \) of potential leaders. We’ll attempt to find out \( k \) processes with maximum ids.

**Algorithm:**

{for **red** process \( i \)}

Initially \( \text{Leader}[] \) is empty

Send out token \(<i>\) and \( \text{Leader}[0] := i; \)

**do** \( \text{token}<j> \) && (there are less than \( k \) elements in \( \text{Leader}[] \)) **do**

Insert \( j \) in \( \text{Leader}[] \) so that \( \text{Leader}[] \) is sorted in a decreasing manner (insertion sort);

Forward \( \text{token}<j>; \)

\( \text{token}<j> \) && (there is at least one element \( e \) in \( \text{Leader}[] \) st. \( j>e \)) **do**

Insert \( j \) in \( \text{Leader}[] \) appropriately

// The element \( e \) and higher elements will be pushed one place right in the array and the last element will be removed.

\( \text{token}<j> \) && (there is no element \( e \) in \( \text{Leader}[] \) st. \( j>e \)) **do**

Forward \( \text{token}<j>; \)

//It means process \( i \) has already seen \( k \) process-ids higher than \( j \)
// It’s unnecessary to forward \( \text{token}<j> \) further
// We’re doing it just to ensure termination

\( \text{token}<i> \) **do**  turn **blue**

// Note that the token is killed here

**od**
{for blue process $i$}

**do** token$<j>$ do forward token$<j>$ **od**

**Termination:** Eventually all processes are blue and hence all tokens are killed. Hence the algorithm terminates.

**Correctness:** When the algorithm stops, it’s easy to see that all the processes has seen all $n$ process-ids and chosen the $k$ maximum ids in their Leader[] array. So those $k$ processes are the leaders and every process knows the $k$ leader-ids correctly.

**Complexity:** There are $n$ tokens and they travel $n$ edges (round the ring). So the message complexity is independent of $k$.

**Problem 3:**

We should utilize the fact that the network topology is a tree and use the usual probe-echo method. (Thanks to Terrance Mason for the following pseudocode)

```plaintext
program leader of tree (probe-echo-timestamp)
define n: integer {number of neighbors}
c, d: integer
steps: integer (number of steps down the path to break ties)
initially parent = i, c,d = 0; ts = null; steps = 0;

{for the initiator}
send probes to each neighbor; d = number of neighbors; steps = 1;
**do**
    d != 0 ^ echo _ d=d-1;
**od**

{for non-initiator process $i$}
do
probes =i ^ c=0 _ c:=1; parent:=sender; ts = probe.ts; steps = probe.steps
    if i is a leaf 
      send echo to parent
    else
      send probes to non-parent neighbors;
      d:= number of non-parent neighbors
    fi
echo _ d:=d-1;
probes =i ^ probe.steps<steps _
    send echo; parent:=sender;
    steps:=probe.steps //new parent probes
```
if i is a leaf
   send echo to parent
i not a leaf
   send probes to non-parent neighbors;
d:= number of non-parent neighbors; //new parent
fi

probe^parent != i ^probe.steps=steps
   flip coin to find winner tree
if(loser of coin flip) //give up growth and let other tree grow
   send echo; parent:=sender;
   steps:=probe.steps //new parent probes
   send probes to non-parent neighbors;
d:= number of non-parent neighbors; //new parent
fi

   c=1^d= 0
   send echo;
od