一、实验内容

1、 基于已有代码，实现生成树运行机制，对于给定拓扑(four_node_ring.py)，计算输出相应状态下的生成树拓扑

2、 自己构造一个不少于 7 个节点，冗余链路不少于 2 条的拓扑，节点和端口的命名规则可参考 four_node_ring.py，使用 stp 程序计算输出生成树拓扑

二、实验流程

1、 基于已有代码，实现生成树运行机制

(1) 修改 `static void stp_handle_config_packet(stp_t *stp, stp_port_t *p, struct stp_config *config)`（stp.c）：

本端口处理收到 stp_config 时：

首先比较本端口和发送端口的优先级。

倘若优先级更高则进行以下处理：

更新本端口的 config

更新本端口对应的本节点的 config

更新本节点其他端口的 config

若本节点不再是根节点，则停止本节点的计时器，并将 config 通过指定端口发出。

```
static void stp_handle_config_packet(stp_t *stp, stp_port_t *p, struct
stp_config *config)
{
    // TODO: handle config packet here
    fprintf(stdout, "TODO: handle config packet here.\n");
    if (recv_has_higher_pirority(p, config)) {
        update_port_config(p, config);
        update_root(stp, config);
        update_other_ports(stp);
        if(!stp_is_root_switch(stp)) {
            stp_stop_timer(&stp->hello_timer);
            stp_send_config(stp);
        }
    }
}
```

(2) 补充 `int recv_has_higher_pirority(stp_port_t * p, struct stp_config *config)`（stp.c）：

该函数的作用是比较本端口和发送端口的 config 优先级，如果本端口优先级更高返回 0，否则返回 1。首先比较节点 ID，其次比较到根节点的开销，再比较上一跳节点的 ID，最后比较端口

ID。所有的比较均为小的值优先级更高。

```
int recv_has_higher_pirority(stp_port_t * p, struct stp_config *config)
{
    if (p->designated_root != ntohll(config->root_id)) {
        if(p->designated_root < ntohll(config->root_id)) {
            return 0;
        } else {
            return 1;
        }
    } else if (p->designated_cost != ntohl(config->root_path_cost)) {
        if (p->designated_cost < ntohl(config->root_path_cost)) {
            return 0;
        } else {
            return 1;
        }
    } else if (p->designated_switch != ntohll(config->switch_id)) {
        if (p->designated_switch < ntohll(config->switch_id)) {
            return 0;
        } else {
            return 1;
        }
    } else if (p->designated_port != ntohs(config->port_id)) {
        if (p->designated_port < ntohs(config->port_id)) {
            return 0;
        } else {
            return 1;
        }
    } else {
        return 1;
    }
}
```

(3) 补充int compare_ports_pirority(stp_port_t * p1, stp_port_t * p2)（stp.c）:

该函数的作用是比较本节点中不同端口的优先级，代码实现与recv_has_higher_pirority函数类似，唯一的不同是不需要进行字节序的转换。

```
int compare_ports_pirority(stp_port_t * p1, stp_port_t * p2) {
    if (p1->designated_root != p2->designated_root) {
        if(p1->designated_root < p2->designated_root) {
            return 0;
        } else {
            return 1;
        }
    } else if (p1->designated_cost != p2->designated_cost) {
        if (p1->designated_cost < p2->designated_cost) {
            return 0;
        } else {
            return 1;
        }
    } else if (p1->designated_switch != p2->designated_switch) {
        if (p1->designated_switch < p2->designated_switch) {
            return 0;
        } else {
            return 1;
        }
    } else if (p1->designated_port != p2->designated_port) {
        if (p1->designated_port < p2->designated_port) {
            return 0;
        } else {
            return 1;
        }
    } else {
        return 1;
    }
}
```

(4) 补充void update_port_config(stp_port_t *p, struct stp_config *config)(stp.c)：

该函数的作用是根据接收端口发送config的值，来更新本端口的config。（同时本端口会变为非指定端口）

```
void update_port_config(stp_port_t *p, struct stp_config *config) {
    p->designated_root = ntohll(config->root_id);
    p->designated_switch = ntohll(config->switch_id);
    p->designated_port = ntohs(config->port_id);
    p->designated_cost = ntohl(config->root_path_cost);
}
```

(5) 补充void update_root(stp_t *stp, struct stp_config *config)（stp.c)：

该函数的作用是更新节点的状态。

```
void update_root(stp_t *stp, struct stp_config *config) {
    stp_port_t * non_designated_ports[STP_MAX_PORTS];
    int num_non_ports = 0;
    for (int i =0 ; i < stp->nports; i++) {
        if(!stp_port_is_designated(&stp->ports[i])) {
            non_designated_ports[num_non_ports] = &stp->ports[i];
            num_non_ports ++;
        }
    }
    int judge = 0;
    if (num_non_ports == 1) {
        stp->root_port = non_designated_ports[0];
    } else {
        for(int i = 0; i < num_non_ports -1; i++) {
            if (judge == 0) {
                if(!compare_ports_pirority(non_designated_ports[i],
non_designated_ports[i+1])) {
                    stp->root_port = non_designated_ports[i];
                    judge = 1;
                }
            } else {
                if(!compare_ports_pirority(non_designated_ports[i],
stp->root_port)) {
                    stp->root_port = non_designated_ports[i];
                }
            }
        }
    }
    if (stp->root_port == NULL) {
        stp->designated_root = stp->switch_id;
        stp->root_path_cost = 0;
    } else {
        stp->designated_root = stp->root_port->designated_root;
        stp->root_path_cost    =    stp->root_port->designated_cost    +
stp->root_port->path_cost;
    }
}
```

上述代码的逻辑为：

遍历所有端口，满足如下条件的为根端口：

该端口是非指定端口

该端口的优先级要高于所有剩余非指定端口(①)

如果不存在根端口，则该节点为根节点。

否则，选择通过 root_port 连接到根节点，更新节点状态为：

```
        stp->root_port = root_port
        stp->designate_root = root_port->designated_root
        stp->root_path_cost = root_port->designated_cost + root_port->path_cost
```

(6) 补充 void update_other_ports(stp_t *stp)（stp.c）：

该函数的作用是更新节点中其他端口的状态。对于所有指定端口，更新其认为的根节点和路径开销。另外，如果一个端口为非指定端口，且其 config 较网段内其他端口优先级更高，那么该端口成为指定端口。

```
void update_other_ports(stp_t *stp) {
    for (int i =0 ; i < stp->nports; i++) {
        if(stp_port_is_designated(&stp->ports[i])) {
            stp->ports[i].designated_root = stp->designated_root;
            stp->ports[i].designated_cost = stp->root_path_cost;
        }
    }
}
```

(7) 修改 void handle_packet(iface_info_t *iface, char *packet, int len)

```
void handle_packet(iface_info_t *iface, char *packet, int len)
{
    struct ether_header *eh = (struct ether_header *)packet;
    if (memcmp(eh->ether_dhost, eth_stp_addr, sizeof(*eth_stp_addr))) {
        // fprintf(stdout, "TODO: received non-stp packet, forward it.\n");
        log(DEBUG, "received non-stp packet, ignore it.");
        return ;
    }
    stp_port_handle_packet(iface->port, packet, len);
    free(packet);
}
```

2、 对于给定拓扑(four_node_ring.py)，计算输出相应状态下的生成树拓扑

(1) 执行命令 make，生成可执行程序 stp：

```
wasder@WASDER:~/exp1/3-stp$ make
```

(2) 运行拓扑文件 four_node_ring.py，启动 Mininet 网络：

```
wasder@WASDER:~/exp1/3-stp$ sudo python3 four_node_ring.py
```

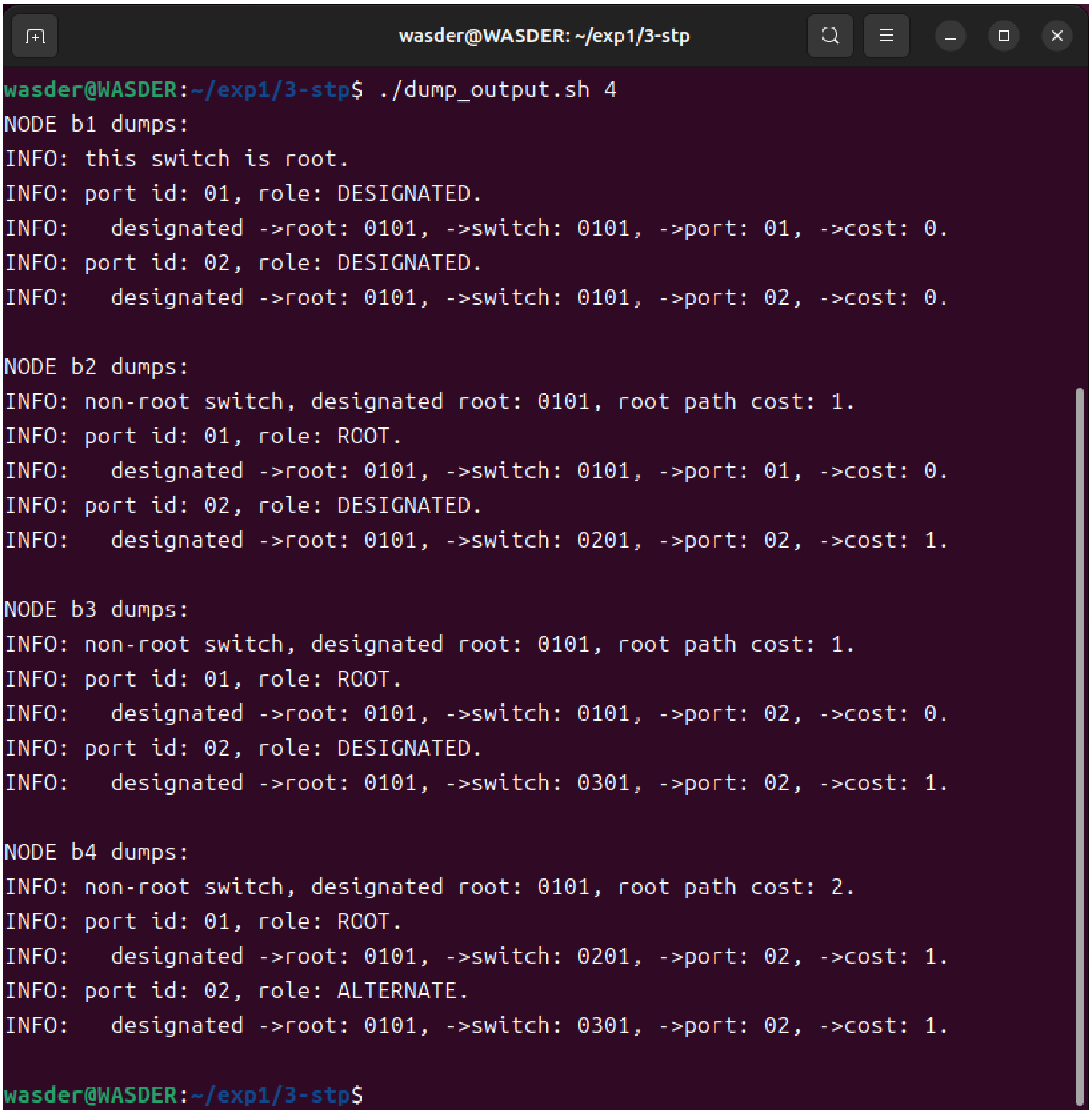
(3) 打开 4 个节点的终端窗口，分别运行 stp 程序，将输出重定向到 b*-output.txt 文件（以 b1 为例）：

```
mininet> xterm b1
b1# ./stp > b1-output.txt 2>&1
```

(4) 等待一段时间（大概 30 秒钟）后，打开新的终端窗口，执行如下命令（以 b1 为例）强制所有 stp 程序输出最终状态并退出：

```
mininet> xterm b1
b1# pkill -SIGTERM stp
```

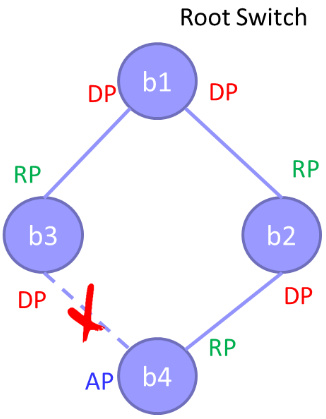
(5) 执行 dump_output.sh 脚本，输出个 4 个节点的状态：



```
wasder@WASDER:~/exp1/3-stp$ ./dump_output.sh 4
NODE b1 dumps:
INFO: this switch is root.
INFO: port id: 01, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0101, ->port: 01, ->cost: 0.
INFO: port id: 02, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0101, ->port: 02, ->cost: 0.

NODE b2 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 1.
INFO: port id: 01, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0101, ->port: 01, ->cost: 0.
INFO: port id: 02, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0201, ->port: 02, ->cost: 1.

NODE b3 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 1.
INFO: port id: 01, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0101, ->port: 02, ->cost: 0.
INFO: port id: 02, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0301, ->port: 02, ->cost: 1.

NODE b4 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 2.
INFO: port id: 01, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0201, ->port: 02, ->cost: 1.
INFO: port id: 02, role: ALTERNATE.
INFO:   designated ->root: 0101, ->switch: 0301, ->port: 02, ->cost: 1.

wasder@WASDER:~/exp1/3-stp$
```

可以看出该 4 节点的测试成功。

(6) 可视化 4 节点 STP 生成的结果：



3、 自己构造一个不少于 7 个节点，冗余链路不少于 2 条的拓扑，节点和端口的命名规则可参考 four_node_ring.py，使用 stp 程序计算输出生成树拓扑

(1) 建立 four_node_ring.py 的副本 seven_node_ring.py：

```
wasder@WASDER:~/exp1/3-stp$ cp four_node_ring.py seven_node_ring.py
```

(2) 根据所需拓扑修改 seven_node_ring.py：

```
class RingTopo(Topo):
    def build(self):
        b1 = self.addHost('b1')
        b2 = self.addHost('b2')
        b3 = self.addHost('b3')
        b4 = self.addHost('b4')
        b5 = self.addHost('b5')
        b6 = self.addHost('b6')
        b7 = self.addHost('b7')

        self.addLink(b1, b2)
        self.addLink(b1, b3)
        self.addLink(b2, b4)
        self.addLink(b3, b5)
        self.addLink(b4, b6)
        self.addLink(b5, b7)
        self.addLink(b4, b5)
        self.addLink(b6, b7)
```

(3) 运行拓扑文件 seven_node_ring.py，启动 Mininet 网络：

```
wasder@WASDER:~/exp1/3-stp$ sudo python3 seven_node_ring.py
```

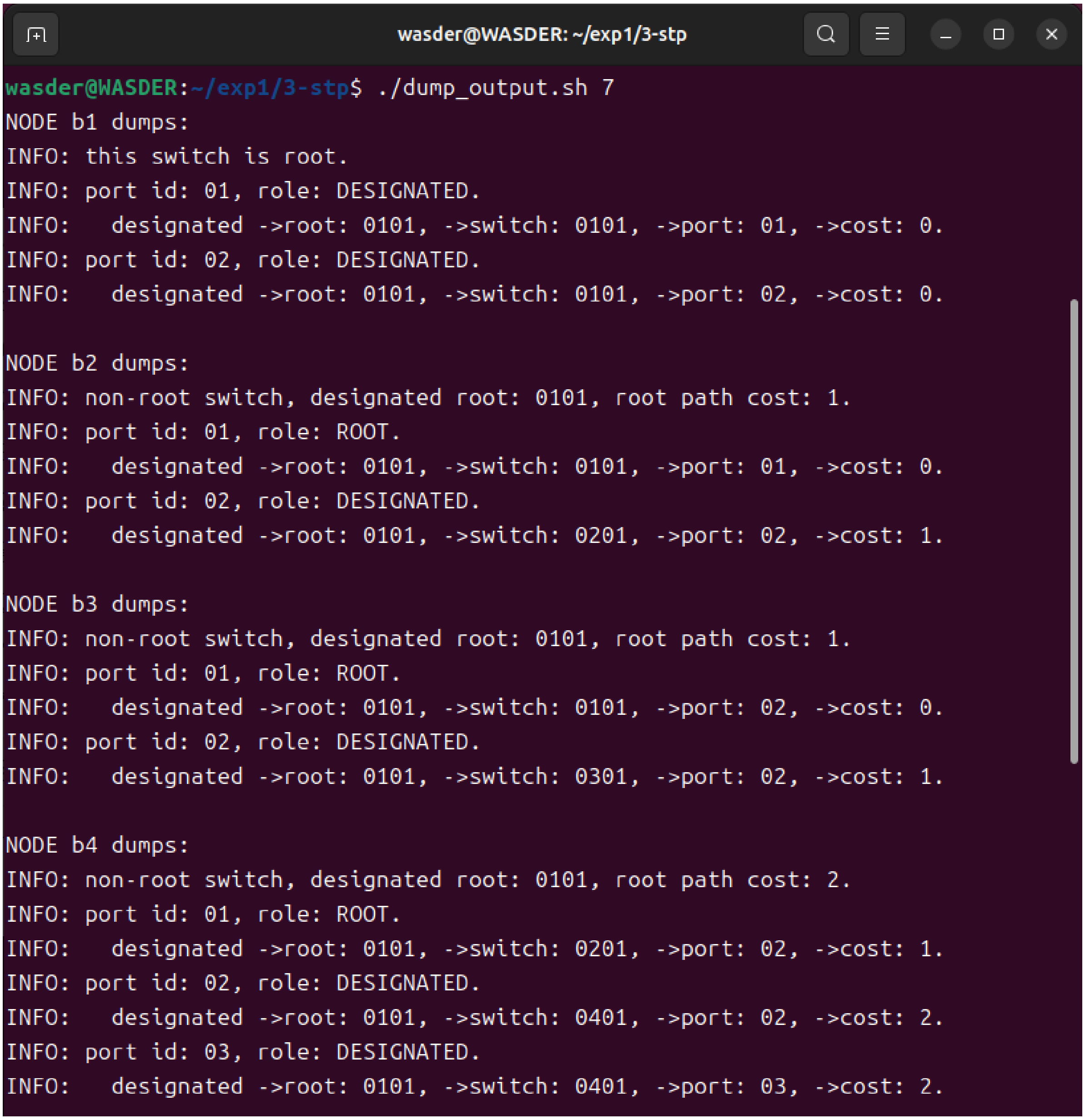
(4) 打开 7 个节点的终端窗口，分别运行 stp 程序，将输出重定向到 b*-output.txt 文件（以 b1 为例）：

```
mininet> xterm b1
b1# ./stp > b1-output.txt 2>&1
```

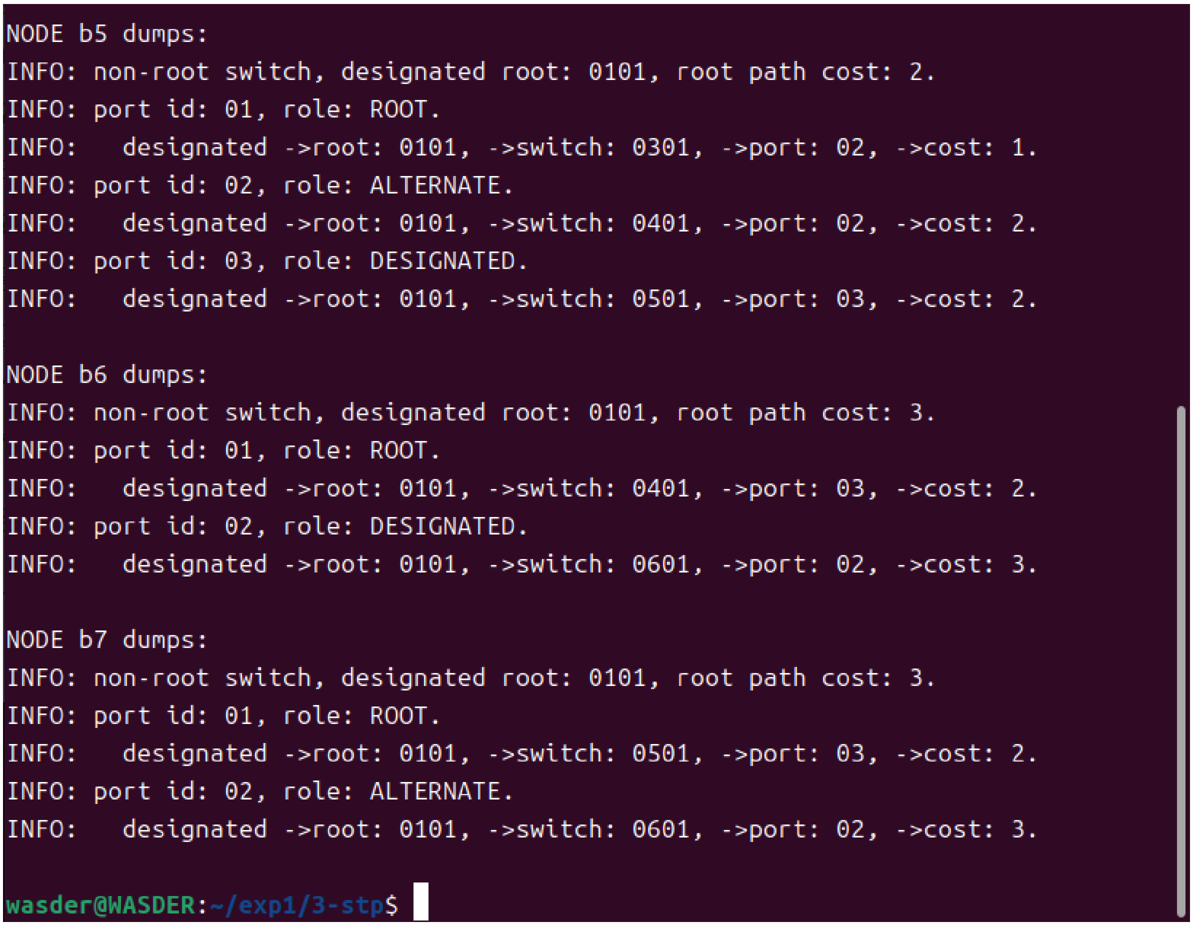
(5) 等待一段时间（大概 30 秒钟）后，打开新的终端窗口，执行如下命令（以 b1 为例）强制所有 stp 程序输出最终状态并退出：

```
mininet> xterm b1
b1# pkill -SIGTERM stp
```

(6) 执行 dump_output.sh 脚本，输出个 7 个节点的状态：



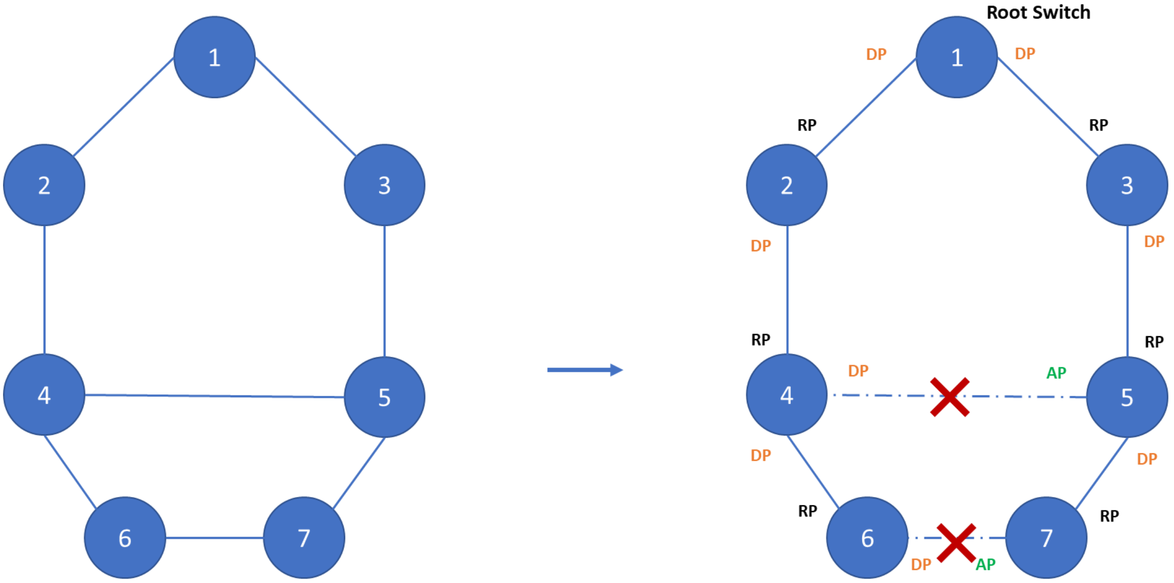
```
wasder@WASDER:~/exp1/3-stp$ ./dump_output.sh 7
NODE b1 dumps:
INFO: this switch is root.
INFO: port id: 01, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0101, ->port: 01, ->cost: 0.
INFO: port id: 02, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0101, ->port: 02, ->cost: 0.

NODE b2 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 1.
INFO: port id: 01, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0101, ->port: 01, ->cost: 0.
INFO: port id: 02, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0201, ->port: 02, ->cost: 1.

NODE b3 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 1.
INFO: port id: 01, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0101, ->port: 02, ->cost: 0.
INFO: port id: 02, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0301, ->port: 02, ->cost: 1.

NODE b4 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 2.
INFO: port id: 01, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0201, ->port: 02, ->cost: 1.
INFO: port id: 02, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0401, ->port: 02, ->cost: 2.
INFO: port id: 03, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0401, ->port: 03, ->cost: 2.
```



```
NODE b5 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 2.
INFO: port id: 01, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0301, ->port: 02, ->cost: 1.
INFO: port id: 02, role: ALTERNATE.
INFO:   designated ->root: 0101, ->switch: 0401, ->port: 02, ->cost: 2.
INFO: port id: 03, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0501, ->port: 03, ->cost: 2.

NODE b6 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 3.
INFO: port id: 01, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0401, ->port: 03, ->cost: 2.
INFO: port id: 02, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0601, ->port: 02, ->cost: 3.

NODE b7 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 3.
INFO: port id: 01, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0501, ->port: 03, ->cost: 2.
INFO: port id: 02, role: ALTERNATE.
INFO:   designated ->root: 0101, ->switch: 0601, ->port: 02, ->cost: 3.

wasder@WASDER:~/exp1/3-stp$ 
```

可以看出该 7 节点的测试成功。

(8) 可视化 7 节点 STP 生成的结果：


1
2
3
4
5
6
7
Root Switch
DP
1
DP
RP
RP
2
3
DP
DP
RP
RP
DP
AP
4
5
DP
DP
RP
RP
6
7
DP
AP