```
-module(node3).
-export([start/1, start/2]).

-define(Stabilize, 1000).
-define(Timeout, 5000).

start(MyKey) ->
    start(MyKey, nil).

start(MyKey, PeerPid) ->
    timer:start(),
    spawn(fun() -> init(MyKey, PeerPid) end).

init(MyKey, PeerPid) ->
    Predecessor = nil,
    {ok, Successor} = connect(MyKey, PeerPid),
    schedule_stabilize(),
    Next = nil, ✓
    Store = storage:create(), ✓
    node(MyKey, Predecessor, Successor, Next, Store).

connect(MyKey, nil) ->
    {ok, { MyKey ✓ , nil ✓ , self() ✓ }};
connect(_, PeerPid) ->
    Qref = make_ref(),
    PeerPid ! {key, Qref, self()},
    receive
        {Qref, Skey} ->
            {ok, { Skey ✓ , monit(PeerPid) ✓ , PeerPid ✓ }}
    after ?Timeout ->
        io:format("Timeout: no response from ~w~n", [PeerPid])
    end.

schedule_stabilize() ->
    timer:send_interval(?Stabilize, self(), stabilize).

node(MyKey, Predecessor, Successor, Next, Store) ->
    receive
        {key, Qref, Peer} ->
            Peer ! {Qref, MyKey},
            node(MyKey, Predecessor, Successor, Next, Store);
        {notify, NewPeer} ->
            {NewPredecessor, NewStore} ✓ = notify(NewPeer, MyKey, Predecessor, Store),
            node(MyKey, NewPredecessor, Successor, Next, NewStore);
        {request, Peer} ->
            request(Peer, Predecessor, Successor),
            node(MyKey, Predecessor, Successor, Next, Store);
        {status, Pred, Nx} ->
            {NewSuccessor, NewNext} ✓ = stabilize(Pred, Nx, MyKey, Successor),
            node(MyKey, Predecessor, NewSuccessor, NewNext, Store);
        stabilize ->
            stabilize(Successor),
            node(MyKey, Predecessor, Successor, Next, Store);
        {add, Key, Value, Qref, Client} ->
            Added = add(Key, Value, Qref, Client, MyKey, Predecessor, Successor, Store),
            node(MyKey, Predecessor, Successor, Next, Added);
        {lookup, Key, Qref, Client} ->
            lookup(Key, Qref, Client, MyKey, Predecessor, Successor, Store),
            node(MyKey, Predecessor, Successor, Next, Store);
        {handover, Elements} ->
            NewStore = storage:merge(Store, Elements), ✓
            node(MyKey, Predecessor, Successor, Next, NewStore);
        {'DOWN', Ref, process, _, _} ->
            {NewPred, NewSucc, NewNext} = down(Ref, Predecessor, Successor, Next),
            node(MyKey, NewPred, NewSucc, NewNext, Store);
        stop ->
            ok;
        probe ->
```

```
            create_probe(MyKey, Successor, Store),
            node(MyKey, Predecessor, Successor, Next, Store);
        {probe, MyKey, Nodes, T} ->
            remove_probe(MyKey, Nodes, T),
            node(MyKey, Predecessor, Successor, Next, Store);
        {probe, RefKey, Nodes, T} ->
            forward_probe(MyKey, RefKey, [MyKey|Nodes], T, Successor, Store),
            node(MyKey, Predecessor, Successor, Next, Store);
        Error ->
            io:format("Reception of strange message ~w~n", [Error]),
            node(MyKey, Predecessor, Successor, Next, Store)
   end.

stabilize(Pred, Next, MyKey, Successor) ->
  {Skey, Sref, Spid} = Successor,
  case Pred of
      nil ->
          Spid ! {notify, {MyKey, self()}}, ✓
          {Successor, Next};
      {MyKey, _} ->
          {Successor, Next};
      {Skey, _} ->
          Spid ! {notify, {MyKey, self()}}, ✓
          {Successor, Next};
      {Xkey, Xpid} ->
            case key:between(Xkey, MyKey, Skey) of
                true ->
                    self() ! stabilize, ✓
                    demonit(Sref), ✓
                    {{Xkey, monit(Xpid), Xpid}, {Skey, Spid}}; ✓
                false ->
                    Spid ! {notify, {MyKey, self()}}, ✓
                    {Successor, Next} ✓
            end
    end.

stabilize( {_, _, Spid} ✓ ) ->
    Spid ! {request, self()}.

request(Peer, Predecessor, {Skey, _, Spid} ✓ ) ->
    case Predecessor of
        nil ->
            Peer ! {status, nil, {Skey, Spid}};
        {Pkey, _, Ppid} ✓ ->
            Peer ! {status, {Pkey, Ppid}, {Skey, Spid}}
    end.

notify({Nkey, Npid}, MyKey, Predecessor, Store) ->
    case Predecessor of
        nil ->
            Keep = handover(Store, MyKey, Nkey, Npid),
            {{Nkey, monit(Npid), Npid}, Keep}; ✓
        {Pkey, Pref, _} ->
            case key:between(Nkey, Pkey, MyKey) of
                true ->
                    Keep = handover(Store, MyKey, Nkey, Npid), ✓
                    demonit(Pref), ✓
                    {{Nkey, monit(Npid), Npid}, Keep}; ✓
                false ->
                {Predecessor, Store} ✓
            end
    end.

add(Key, Value, Qref, Client, _, nil, {_, _, Spid} ✓ , Store) ->
    Spid ! {add, Key, Value, Qref, Client}, ✓
    Store;
add(Key, Value, Qref, Client, MyKey, {Pkey, _, _} ✓ , {_, _, Spid} ✓ , Store) ->
    case key:between( Key ✓ , Pkey ✓ , MyKey ✓ ) of
        true ->
            Added = storage:add(Key, Value, Store), ✓
            Client ! {Qref, ok},
            Added;
        false ->
            Spid ! {add, Key, Value, Qref, Client}, ✓
            Store
    end.

lookup(Key, Qref, Client, _, nil, {_, _, Spid} ✓ , _) ->
    Spid ! {lookup, Key, Qref, Client}; ✓
lookup(Key, Qref, Client, MyKey, {Pkey, _, _} ✓ , {_, _, Spid} ✓ , Store) ->
    case key:between( Key ✓ , Pkey ✓ , MyKey ✓ ) of
        true ->
            Result = storage:lookup(Key, Store), ✓
            Client ! {Qref, Result};
        false ->
            Spid ! {lookup, Key, Qref, Client} ✓
    end.

handover(Store, MyKey, Nkey, Npid) ->
    {Keep, Leave} = storage:split(MyKey, Nkey, Store), ✓
    Npid ! {handover, Leave},
    Keep.

monit(Pid) ->
    erlang:monitor(process, Pid).

demonit(nil) ->
    ok;
demonit(MonitorRef) ->
    erlang:demonitor(MonitorRef, [flush]).

down(Ref, {_, Ref, _}, Successor, Next) ->
    {nil, Successor, Next};
down(Ref, Predecessor, {_, Ref, _}, {Nkey, Npid}) ->
    self() ! stabilize, ✓
    {Predecessor, {Nkey, monit(Npid), Npid} ✓ , nil}.

create_probe(MyKey, {_, _, Spid} ✓ , Store) ->
    Spid ! {probe, MyKey, [MyKey], erlang:monotonic_time()},
    io:format("Node ~w created probe -> Store: ~w~n", [MyKey, Store]).

remove_probe(MyKey, Nodes, T) ->
    T2 = erlang:monotonic_time(),
    Time = erlang:convert_time_unit(T2-T, native, microsecond),
    io:format("Node ~w received probe after ~w us -> Ring: ~w~n", [MyKey, Time, Nodes]).
```

```
forward_probe(MyKey, RefKey, Nodes, T, {_, _, Spid} ✓ , Store) ->
    Spid ! {probe, RefKey, Nodes, T},
    io:format("Node ~w forwarded probe started by node ~w -> Store: ~w~n", [MyKey, RefKey, Store]).
```