1 Elimination Stack

For simplicity, we assume a value $\perp$, which is different from NULL and any other value the stack can store. Since NULL is used as a legit return value, representing the value of POP operation (when exchanging values using the elimination array), NULL can not be used to represent an initialization value, different then any stack value. The same holds for a Node, since a NULL node represent an empty stack, the value $\perp$ is used to distinguish between initialization value and empty stack.

For simplicity, we split the RECOVER routine into sub-routines, based on which operation (PUSH, POP, EXCHANGE) is pending, or needs to be recover. This can be concluded easily by the type of record stored in *Announce*[*pid*] (ExInfo or OpInfo), thus there is no need to explicitly know where exactly in the code the crash took place. Also, the RECOVER routine returns FAIL in case the last pending operation did not took affect (no linearization point), nor it will take in any future run. In such case, the user has the option to either re-invoke the operation, or to skip it, depends on the needs and circumstances of the specific use of the data structure.

The given implementation ignores the log of failures and successes of the exchange routine when recovering. That is, in case of a crash during an EXCHANGE, a process is able to recover the EXCHANGE routine, however, the log of successes and failures is not update, since it might be the process already updated it. In addition, in case of a FAIL response, we do not know whether the time limit (timeout) was reached, or that the process simply crashed earlier in the routine without completing it. The given implementation can be expanded to also consider the log. Nonetheless, for ease of presentation we do not handle the log in case of a crash. Assuming crash events are rare, the log still gives a roughly good approximation to the number of failures and successes, thus our approach might be useful in practice.

1.1 A Lock-Free Exchanger

An exchanger object supports the EXCHANGE procedure, which allows

```
Type Node {
    T value
    int popby
    Node *next
}

Type OpInfo{                ▷ subtype of Info
    {PUSH, POP} optype
    Node *curr
    boolean done
}

Type ExInfo{                ▷ subtype of Info
    {EMPTY, WAITING, BUSY} state
    T value, result
    ExInfo *partner, *slot
}
```

Figure 1: Type definition

ExInfo *default* - a global static ExInfo object with state = EMPTY

Algorithm 1: T EXCHANGE (ExInfo $*slot$, T *myitem*, long *timeout*)

```
long timeBound := getNanos() + timeout
ExInfo myop := new ExInfo(WAITING, myitem, ⊥, ⊥, slot)
Announce[pid] := myop
while true do
    if getNanos() > timeBound then
        myop.result := TIMEOUT                                   // time limit reached
        return TIMEOUT
    yourop := slot
    switch yourop.state do
        case EMPTY do
            myop.state := WAITING                                 // try to replace default
            myop.partner := ⊥
            if slot.CAS(yourop, myop) then
                while getNanos() < timeBound do
                    yourop := slot
                    if yourop ≠ myop then                         // a collision was done
                        if youop.parnter = myop then              // yourop collide with myop
                            SWITCHPAIR(myop, yourop)
                            slot.CAS(yourop, default)             // release slot
                        return myop.result
                end
                // time limit reached and no process collide with me
                if slot.CAS(myop, default) then                   // try to release slot
                    myop.result := TIMEOUT
                    return TIMEOUT
                else                                              // some process show up
                    yourop := slot
                    if yourop.partner = myop then
                        SWITCHPAIR(myop, yourop)                  // complete the collision
                        slot.CAS(yourop, default)                 // release slot
                    return myop.result
            end
            break
        case WAITING do                                           // some process is waiting in slot
            myop.partner := yourop                                // attempt to replace yourop
            myop.state := BUSY
            if slot.CAS(yourop, myop) then                        // try to collide
                SWITCHPAIR(myop, yourop)                          // complete the collision
                slot.CAS(myop, default)                           // release slot
                return myop.result
            break
        case BUSY do                                              // a collision in progress
            SWITCHPAIR(yourop, yourop.parnter)                    // help to complete the collision
            slot.CAS(yourop, default)                             // release slot
            break
    end
end
```

Algorithm 2: void SwitchPair(ExInfo *first*, ExInfo *second*)

```
// exchange the valus of the two operations
first.result := second.value
second.result := first.value
```

Algorithm 3: T Visit (T *value*, int *range*, long *duration*)

```
// invoke Exchange on a random selected entery in the collision array
int cell := randomNumber(range)
return Exchange(exchanger[cell], value, duration)
```

Algorithm 4: T Exchange-Recover ()

```
ExInfo *myop := Announce[pid]                  // read your last operation record,
ExInfo *slot := myop.slot                      // and the slot on which it act
if myop.state = Waiting then
    // crash while trying to exchange defualt, or waiting for a process to collide
       with me
    yourop := slot
    if yourop = myop then                      // still waiting for a collide
        if slot.CAS(myop, default) then        // try to release slot
            return Fail
        else                                   // some process show up
            yourop := slot
            if yourop.partner = myop then
                SwitchPair(myop, yourop)       // complete the collision
                slot.CAS(yourop, default)      // release slot
            return myop.result
    if yourop.partner = myop then              // yourop collide with myop
        SwitchPair(myop, yourop)               // complete the collision
        slot.CAS(yourop, default)              // release slot
        return myop.result
if myop.state = Busy then
    // crash while trying to collide with myop.partner
    yourop := slot
    if yourop = myop then                      // collide was successful
        SwitchPair(myop, myop.partner)         // complete the collision
        slot.CAS(myop, default)                // release slot
        return myop.result
if myop.result ≠ ⊥ then
    return myop.result                         // collide was successfuly completed
else
    return Fail
```

Figure 2: Elimination Array routines

Algorithm 5: boolean TryPush (Node *new)

```
Node *oldTop := Top
new.next := oldTop
return Top.CAS(oldTop, new)
```

Algorithm 6: boolean Push (T myitem)

```
Node *nd = new Node (myitem)
nd.popby := ⊥
OpInfo data := new OpInfo (Push, nd, false)
while true do
    Announce[pid] := data
    if TryPush(nd) then
        data.done := true
        return true
    range := CalculateRange()
    duration := CalculateDuration()
    othervalue := Visit(myitem, range, duration)
    if othervalue = Null then
        RecordSuccess ()
        return true
    else if othervalue = Timeout then
        RecordFailure ()
end
```

Algorithm 7: boolean Push-Roceover ()

```
OpInfo *data := Announce[pid]
if data.done = true then
    return true
Node *iter := Top
while iter ≠ Null do
    if iter = data.curr then
        data.done := true
        return true
    iter = iter.next
end
if data.curr.popby ≠ ⊥ then
    data.done := true
    return true
return Fail
```

Figure 3: Push routine

Algorithm 8: T TryPop()

```
Node *oldTop := Top
Node *newTop
Announce[pid].curr := oldTop
if oldTop = NULL then
    return EMPTY
newTop := oldTop.next
if Top.CAS(oldTop, newTop) then
    return oldTop
else
    return ⊥
```

Algorithm 9: T Pop ()

```
Node *result
OpInfo data := new OpInfo (POP, ⊥, false)
while true do
    Announce[pid] := data
    result := TryPop()
    if result = EMPTY then
        return EMPTY
    else if result ≠ ⊥ then
        if result.popby.CAS(⊥, pid) then
            return result.value
    range := CalculateRange()
    duration := CalculateDuration()
    othervalue := Visit(NULL, range, duration)
    if othervalue = TIMEOUT then
        RecordFailure ()
    else if othervalue ≠ NULL then
        RecordSuccess ()
        return othervalue
end
```

Algorithm 10: T Pop-Recover()

```
Node *nd := Announce[pid].curr
if nd = EMPTY then
    return EMPTY
if nd = ⊥ then
    return FAIL
if nd.popby ≠ ⊥ then
    if nd.popby = pid then
        return nd.value
    else
        return FAIL
if nd.popby.CAS (NULL, id) then
    return result.value
else
    return FAIL
```


Figure 4: Pop routine