

# 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 PushInfo {                                ▷ subtype of Info
    Node *pushNd
}

Type PopInfo {                                  ▷ subtype of Info
    Node *popNd
}

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*)

---

```

1 long timeBound := getNanos() + timeout
2 ExInfo myop := new ExInfo(WAITING, myitem,  $\perp$ ,  $\perp$ , slot)
3 Announce[pid] := myop
4 while true do
5   if getNanos() > timeBound then
6     | myop.result := TIMEOUT // time limit reached
7     | return TIMEOUT
8   yourop := slot
9   switch yourop.state do
10    case EMPTY
11      | myop.state := WAITING // try to replace default
12      | myop.partner :=  $\perp$ 
13      | if slot.CAS(yourop, myop) then
14        | while getNanos() < timeBound do
15          | | yourop := slot
16          | | if yourop  $\neq$  myop then // a collision was done
17            | | | if yourop.parnter = myop then // yourop collide with myop
18              | | | SWITCHPAIR(myop, yourop)
19              | | | slot.CAS(yourop, default) // release slot
20              | | | return myop.result
21          | | end
22          | | // time limit reached and no process collide with me
23          | | if slot.CAS(myop, default) then // try to release slot
24            | | | myop.result := TIMEOUT
25            | | | return TIMEOUT
26          | | else // some process show up
27            | | | yourop := slot
28            | | | if yourop.partner = myop then
29              | | | | SWITCHPAIR(myop, yourop) // complete the collision
30              | | | | slot.CAS(yourop, default) // release slot
31              | | | | return myop.result
32          | | end
33          | | break
34        | case WAITING // some process is waiting in slot
35          | | myop.partner := yourop // attempt to replace yourop
36          | | myop.state := BUSY
37          | | if slot.CAS(yourop, myop) then // try to collide
38            | | | SWITCHPAIR(myop, yourop) // complete the collision
39            | | | slot.CAS(myop, default) // release slot
40            | | | return myop.result
41          | | break
42        | case BUSY // a collision in progress
43          | | SWITCHPAIR(yourop, yourop.parnter) // help to complete the collision
44          | | slot.CAS(yourop, default) // release slot
45          | | break
46    endsw
47 end

```

---

<b>Algorithm 2:</b> void SWITCHPAIR(ExInfo <i>first</i> , ExInfo <i>second</i> )	
<hr/> // exchange the value of the two operations 47 <i>first.result</i> := <i>second.value</i> 48 <i>second.result</i> := <i>first.value</i> <hr/>	
<b>Algorithm 3:</b> T VISIT (T <i>value</i> , int <i>range</i> , long <i>duration</i> )	
<hr/> // invoke EXCHANGE on a random selected entry in the collision array 49 int <i>cell</i> := randomNumber( <i>range</i> ) 50 <b>return</b> EXCHANGE( <i>exchanger</i> [ <i>cell</i> ], <i>value</i> , <i>duration</i> ) <hr/>	
<b>Algorithm 4:</b> T EXCHANGE-RECOVER ()	
<hr/> 51 ExInfo * <i>myop</i> := Announce[ <i>pid</i> ] // read your last operation record, 52 ExInfo * <i>slot</i> := <i>myop.slot</i> // and the slot on which it act 53 <b>if</b> <i>myop.state</i> = WAITING <b>then</b>   // crash while trying to exchange <i>default</i> , or waiting for a process to collide   with me 54 <i>yourop</i> := <i>slot</i> 55 <b>if</b> <i>yourop</i> = <i>myop</i> <b>then</b> // still waiting for a collide 56   <b>if</b> <i>slot.CAS</i> ( <i>myop</i> , <i>default</i> ) <b>then</b> // try to release slot 57     <b>return</b> FAIL 58   <b>else</b> // some process show up 59     <i>yourop</i> := <i>slot</i> 60     <b>if</b> <i>yourop.partner</i> = <i>myop</i> <b>then</b> 61       SWITCHPAIR( <i>myop</i> , <i>yourop</i> ) // complete the collision 62       <i>slot.CAS</i> ( <i>yourop</i> , <i>default</i> ) // release slot 63     <b>return</b> <i>myop.result</i> 64   <b>if</b> <i>yourop.partner</i> = <i>myop</i> <b>then</b> // <i>yourop</i> collide with <i>myop</i> 65     SWITCHPAIR( <i>myop</i> , <i>yourop</i> ) // complete the collision 66     <i>slot.CAS</i> ( <i>yourop</i> , <i>default</i> ) // release slot 67     <b>return</b> <i>myop.result</i> 68 <b>if</b> <i>myop.state</i> = BUSY <b>then</b>   // crash while trying to collide with <i>myop.partner</i> 69   <i>yourop</i> := <i>slot</i> 70   <b>if</b> <i>yourop</i> = <i>myop</i> <b>then</b> // collide was successful 71     SWITCHPAIR( <i>myop</i> , <i>myop.partner</i> ) // complete the collision 72     <i>slot.CAS</i> ( <i>myop</i> , <i>default</i> ) // release slot 73     <b>return</b> <i>myop.result</i> 74 <b>if</b> <i>myop.result</i> ≠ ⊥ <b>then</b> 75   <b>return</b> <i>myop.result</i> // collide was successfully completed 76 <b>else</b> 77   <b>return</b> FAIL <hr/>	

Figure 2: Elimination Array routines

---

**Algorithm 5:** boolean TRY PUSH (Node \**nd*)

---

```
78 Node *oldTop := Top
79 nd.next := oldTop
80 if Top.CAS(oldTop, nd) then
81   | nd.popby.CAS( $\perp$ , NULL)
82   | return true
83 return false
```

---

---

**Algorithm 6:** boolean PUSH (T *myitem*)

---

```
84 Node *nd = new Node (myitem)
85 nd.popby :=  $\perp$ 
86 PushInfo data := new PushInfo (nd)
87 while true do
88   | Announce[pid] := data
89   | if TRY PUSH(nd) then
90     | return true
91   | range := CalculateRange()
92   | duration := CalculateDuration()
93   | othervalue := VISIT(myitem, range, duration)
94   | if othervalue = NULL then
95     | RecordSuccess ()
96     | return true
97   | else if othervalue = TIMEOUT then
98     | RecordFailure ()
99 end
```

---

---

**Algorithm 7:** boolean PUSH-ROCEOVER ()

---

```
100 Node *nd := Announce[pid].pushNd
101 if nd.popby  $\neq \perp$  then
102   | return true
103 if SEARCH(nd) || nd.popby  $\neq \perp$  then
104   | nd.popby.CAS( $\perp$ , NULL)
105   | return true
106 return FAIL
```

---

---

**Algorithm 8:** boolean SEARCH (Node \**nd*)

---

```
107 Node *iter := Top
108 while iter  $\neq \perp$  do
109   | if iter = nd then
110     | return true
111   | iter := iter.next
112 end
113 return false
```

---

Figure 3: PUSH routine

---

**Algorithm 9: T TRYPOP()**

---

```
114 Node *oldTop := Top
115 Node *newTop
116 Announce[pid].popNd := oldTop
117 if oldTop =  $\perp$  then
118   | return EMPTY
119 newTop := oldTop.next
120 oldTop.popby.CAS( $\perp$ , NULL)
121 if Top.CAS(oldTop, newTop) then
122   | if newTop.popby.CAS(NULL, pid) then
123     | return oldTop.value
124 else
125   | return  $\perp$ 
```

---

---

**Algorithm 10: T POP ()**

---

```
126 Node *result
127 PopInfo data := new PopInfo (Top)
128 while true do
129   | Announce[pid] := data
130   | result := TRYPOP()
131   | if result  $\neq \perp$  then
132     | return result
133   | range := CalculateRange()
134   | duration := CalculateDuration()
135   | othervalue := VISIT(NULL, range, duration)
136   | if othervalue = TIMEOUT then
137     | RecordFailure ()
138   | else if othervalue  $\neq$  NULL then
139     | RecordSuccess ()
140   | return othervalue
141 end
```

---

---

**Algorithm 11: T POP-RECOVER()**

---

```
142 Node *nd := Announce[pid].popNd
143 if nd =  $\perp$  then
144   | return EMPTY
145 if SEARCH(nd) then
146   | return FAIL
147 nd.popby.CAS(NULL, pid)
148 if nd.popby = pid then
149   | return nd.value
150 return FAIL
```

---

Figure 4: POP routine