Resilient Priorirty Queue

Anirban Mitra

anmitra@cs.stonybrook.edu
Department of Computer Science
Stony Brook University

March 30, 2013

Memory Errors

- One or more Memory bits is read differently from last written values
- Causes can be hardware bit corruption, cosmic rays, corruption in path between memory and CPU

Pointer Corruptions

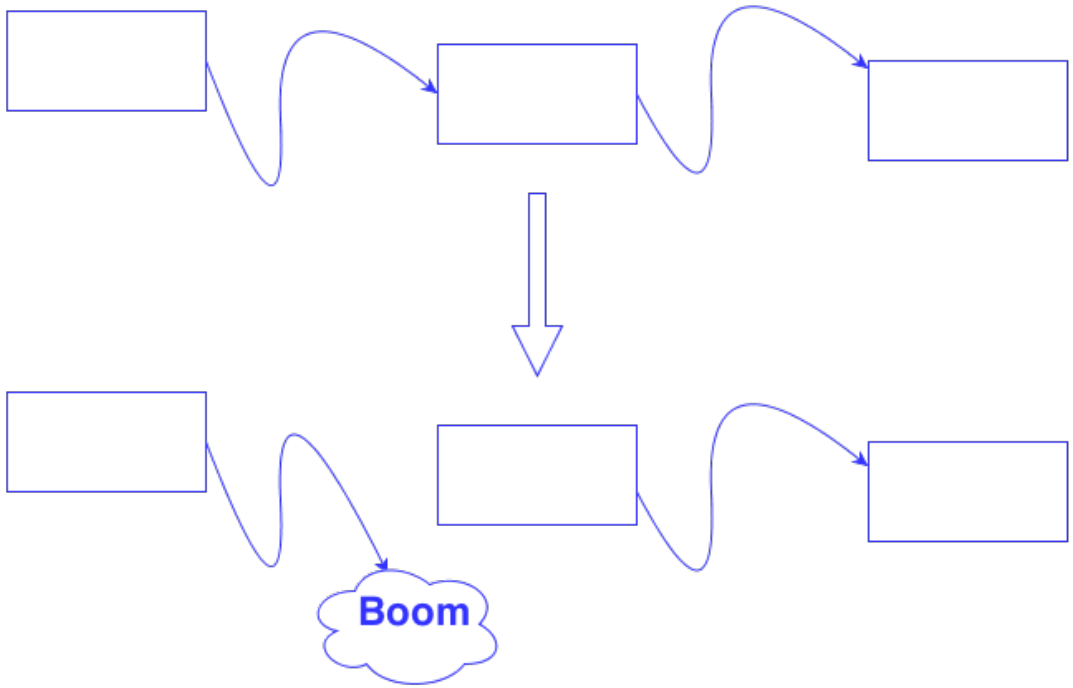


Figure : Even a single memory corruption can be catastrophic. A simple linked list with memory error, entire tail lost due to a single pointer corruption

ECC

- Hardware ECC (Error Correcting Codes) chips can help
- But is expensive in terms of processing time and money
- Even ECC cannot correct every corruption

©Dan Piraro.

Its Not Rare

Mem. size	Mean Time Between Failures
512 MB	2.92 hours
1 GB	1.46 hours
16 GB	5.48 minutes
64 GB	1.37 minutes
1 TB	5.13 seconds

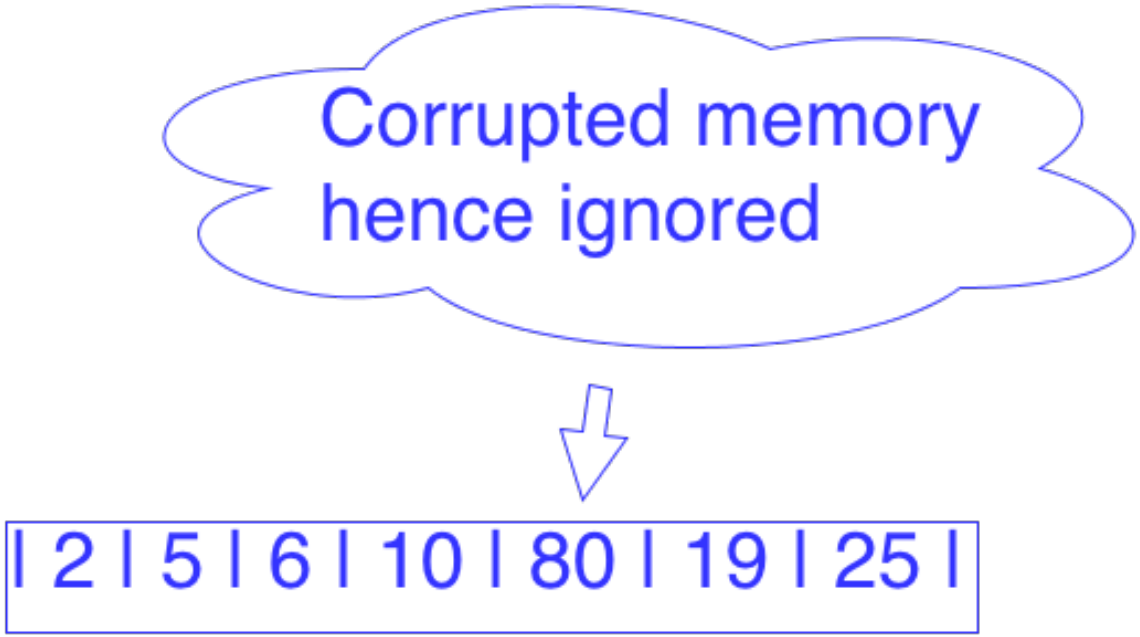
Figure : A field study by Google researchers

Resilient Algorithms

- Make the algorithm and data structure capable of dealing with memory errors
- Design for damage control, minimize the disasters
- Memory model assumes that at maximum δ corruptions throughout the runtime
- δ is an input to the model, known in advance
- Also, assumes that it has some constant amount of relaible memory $P = O(1)$
- Relaible memory cannot get corrupt

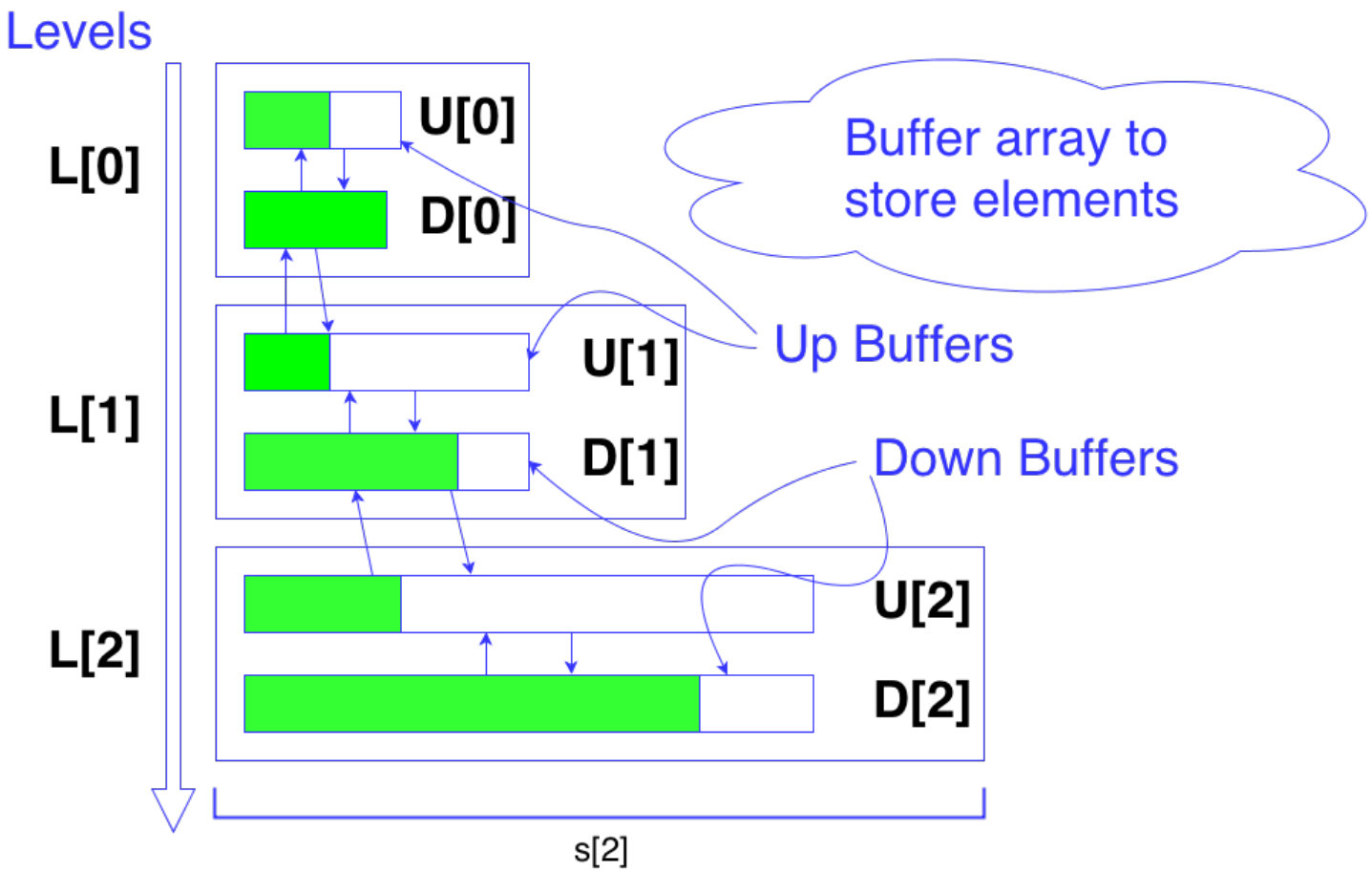
Faithful Ordering

- In resilient sorting, recovery from corruption is expensive
- Hence goal is modified to all un-corrupted items are guaranteed to be correctly sorted
- Order of corrupted items is ignored



Resilient Merging

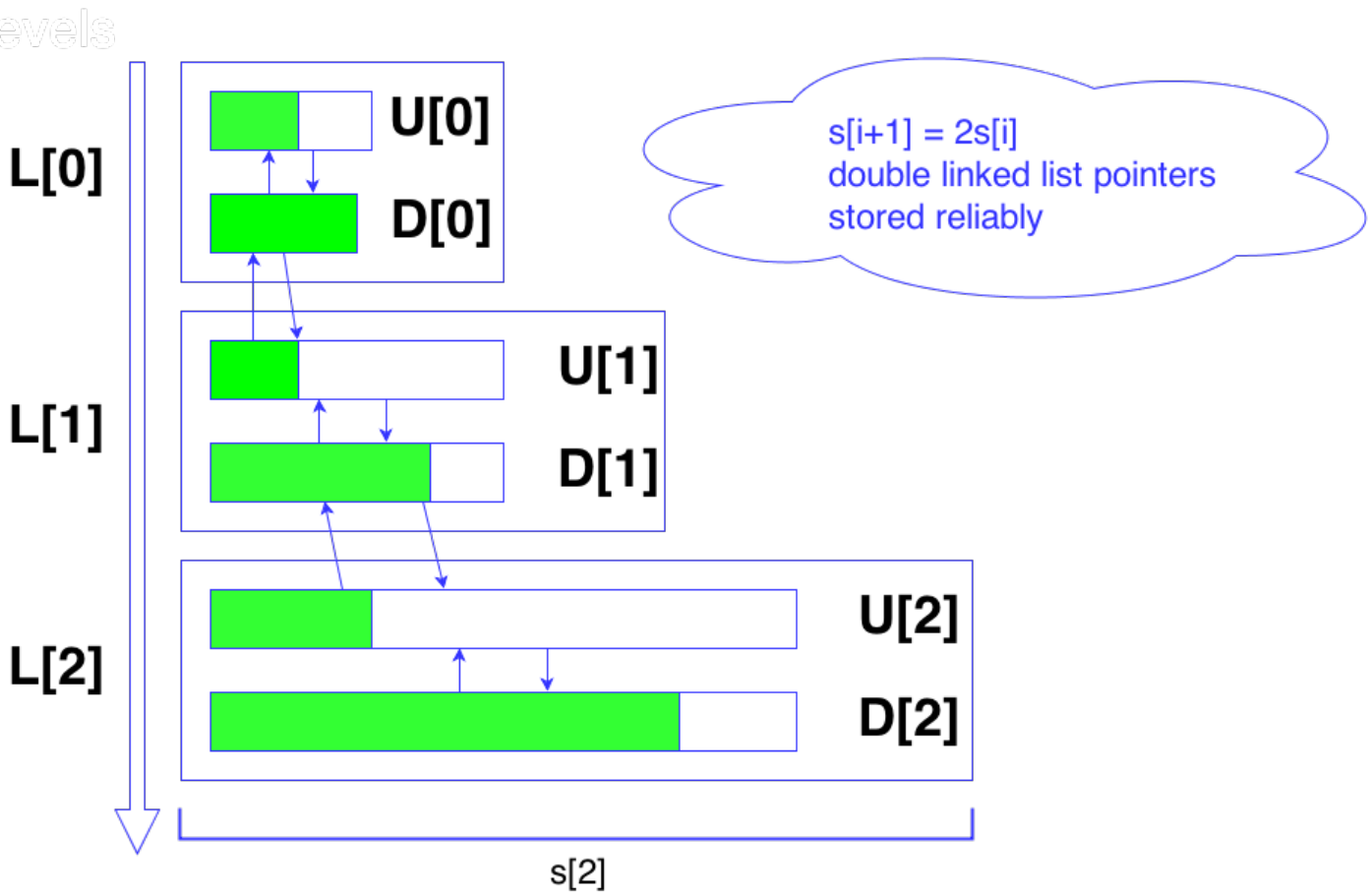
Structure of Resilient Priority Queue



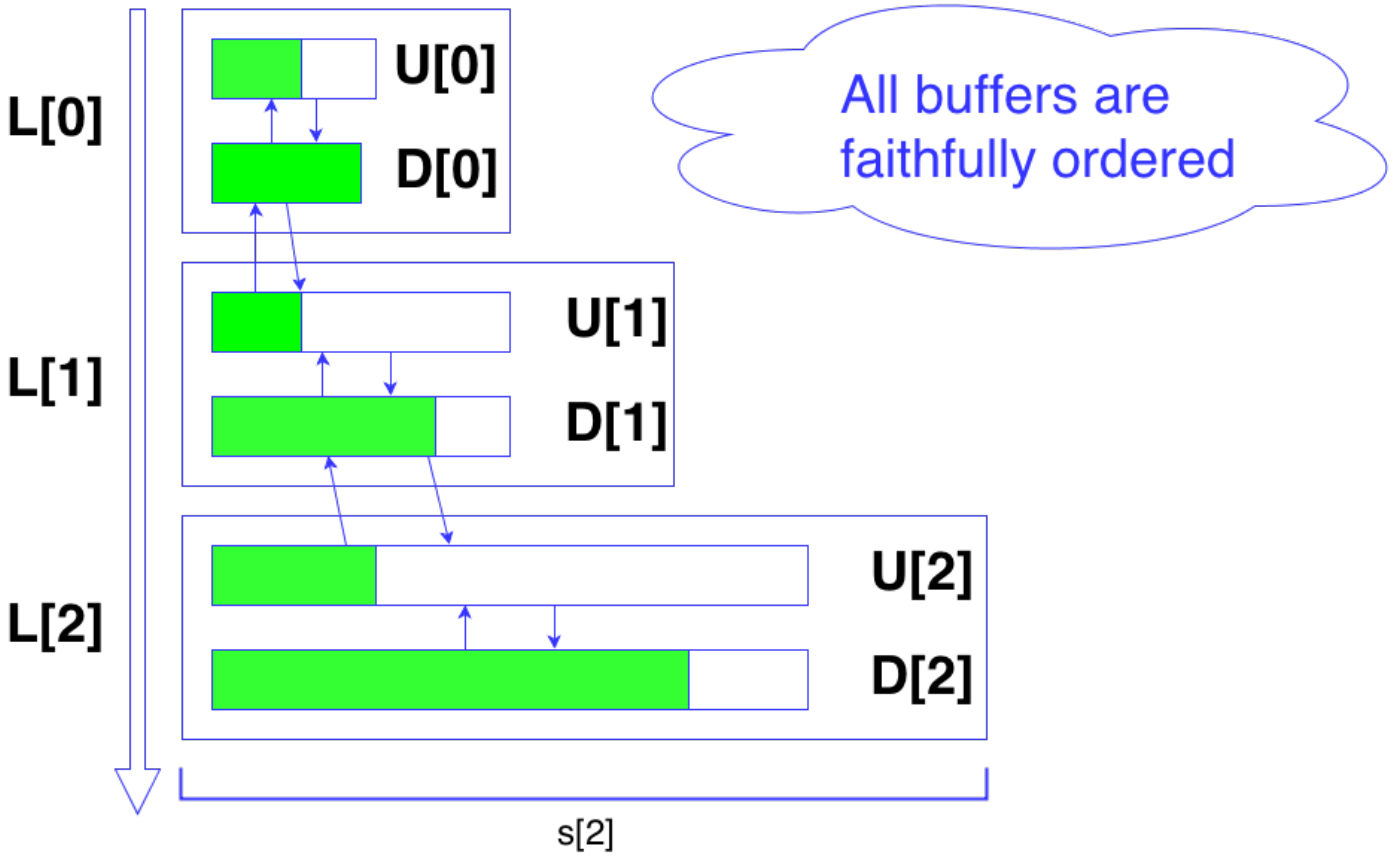
Structure of Resilient Priority Queue

- Intuitively, elements in Up buffer are moving to upper layers
- Elements in Down buffer are moving to lower layers

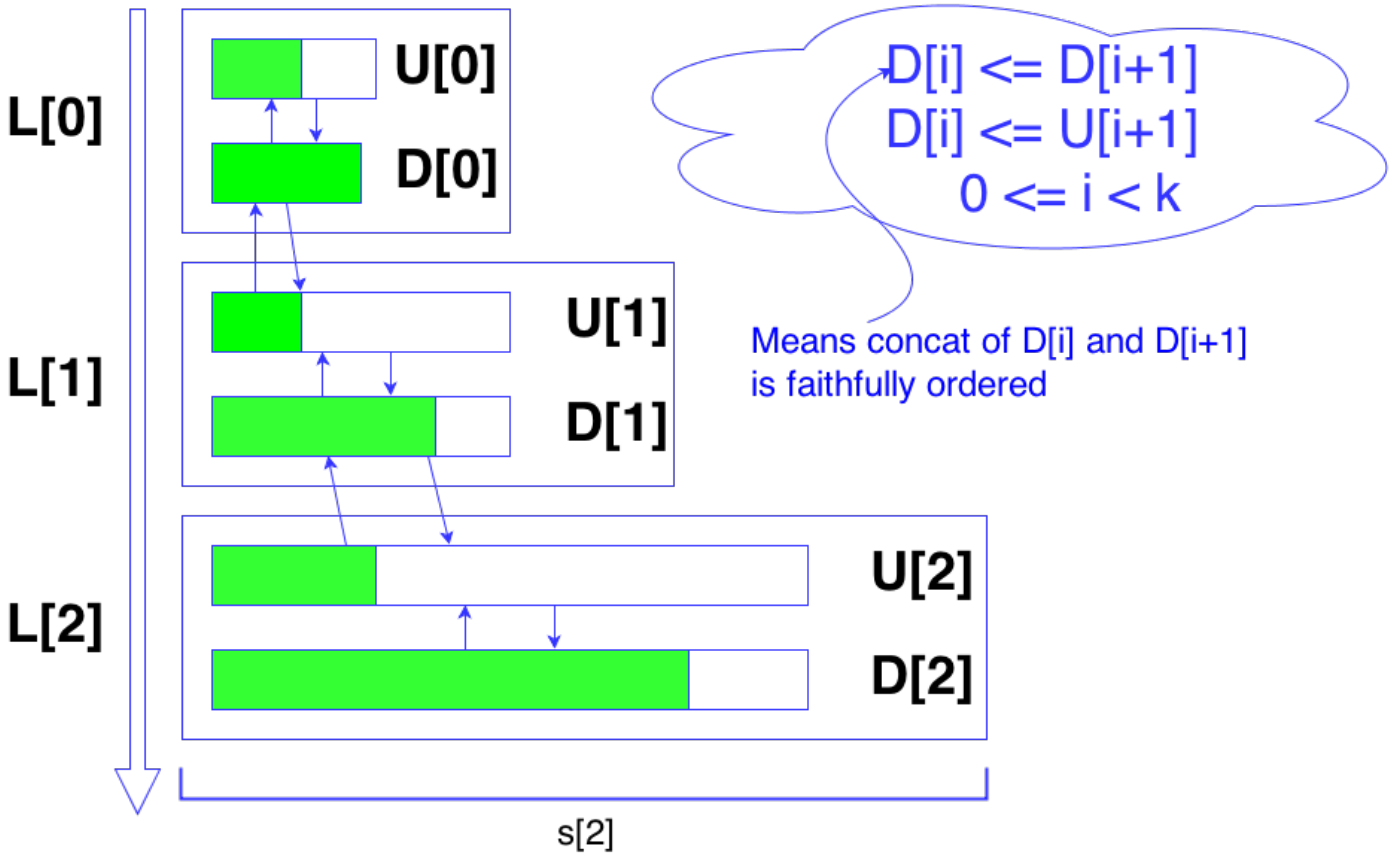
Structure of Resilient Priority Queue



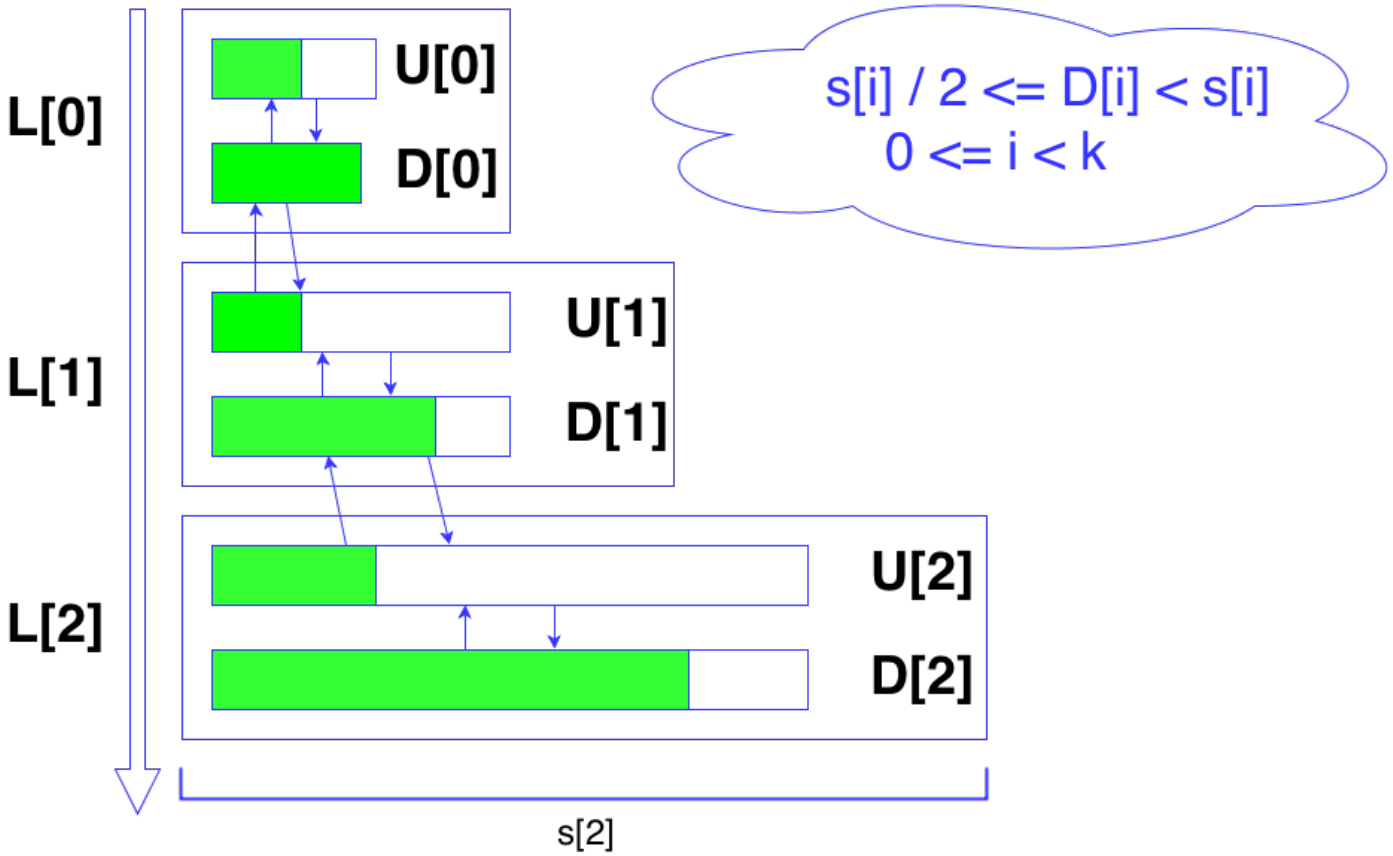
Invariant A


L[0]
L[1]
L[2]
U[0]
D[0]
U[1]
D[1]
U[2]
D[2]
s[2]
All buffers are faithfully ordered

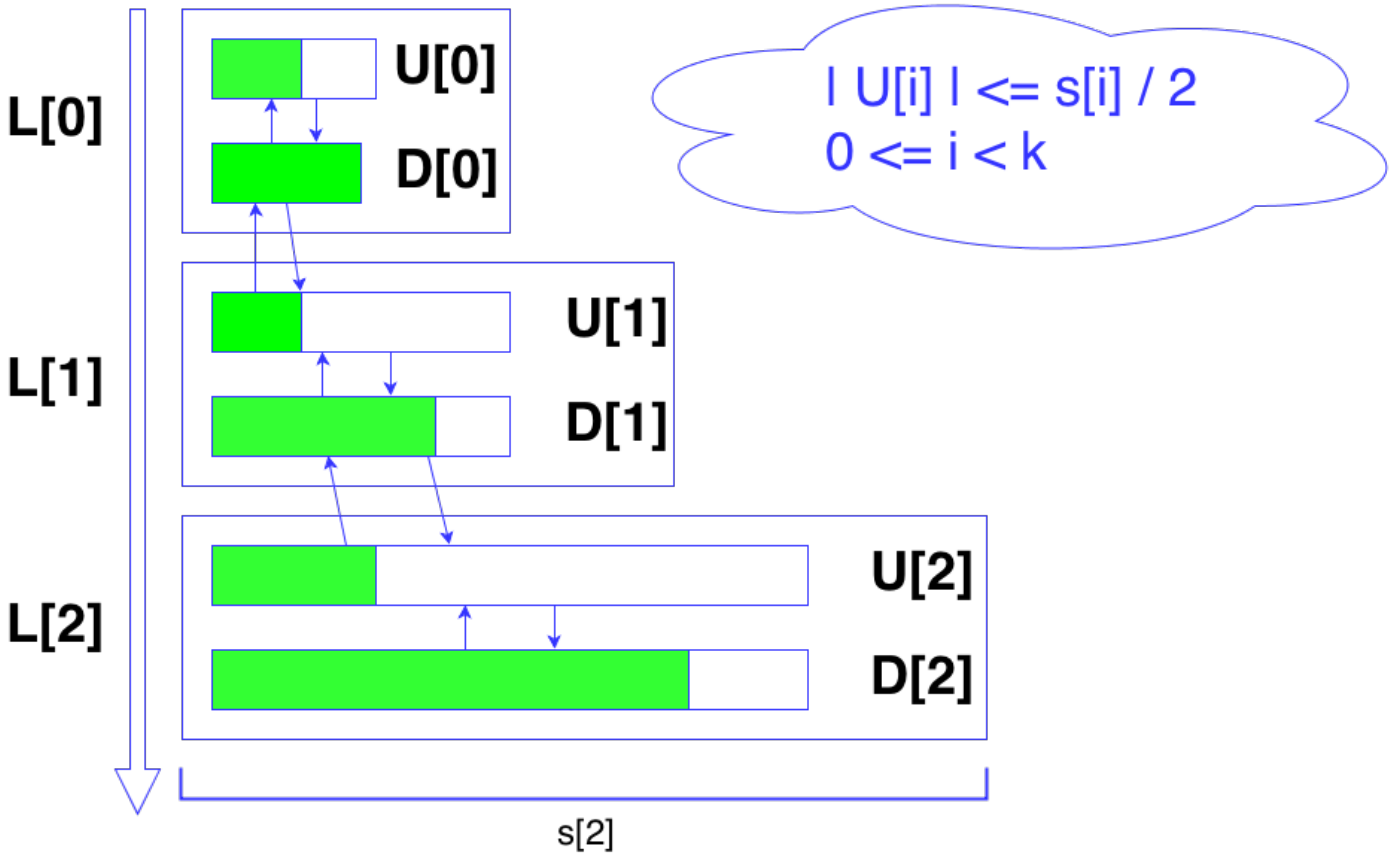
Invariant B


L[0]
L[1]
L[2]
U[0]
D[0]
U[1]
D[1]
U[2]
D[2]
s[2]
D[i] <= D[i+1]
D[i] <= U[i+1]
0 <= i < k
Means concat of D[i] and D[i+1]
is faithfully ordered

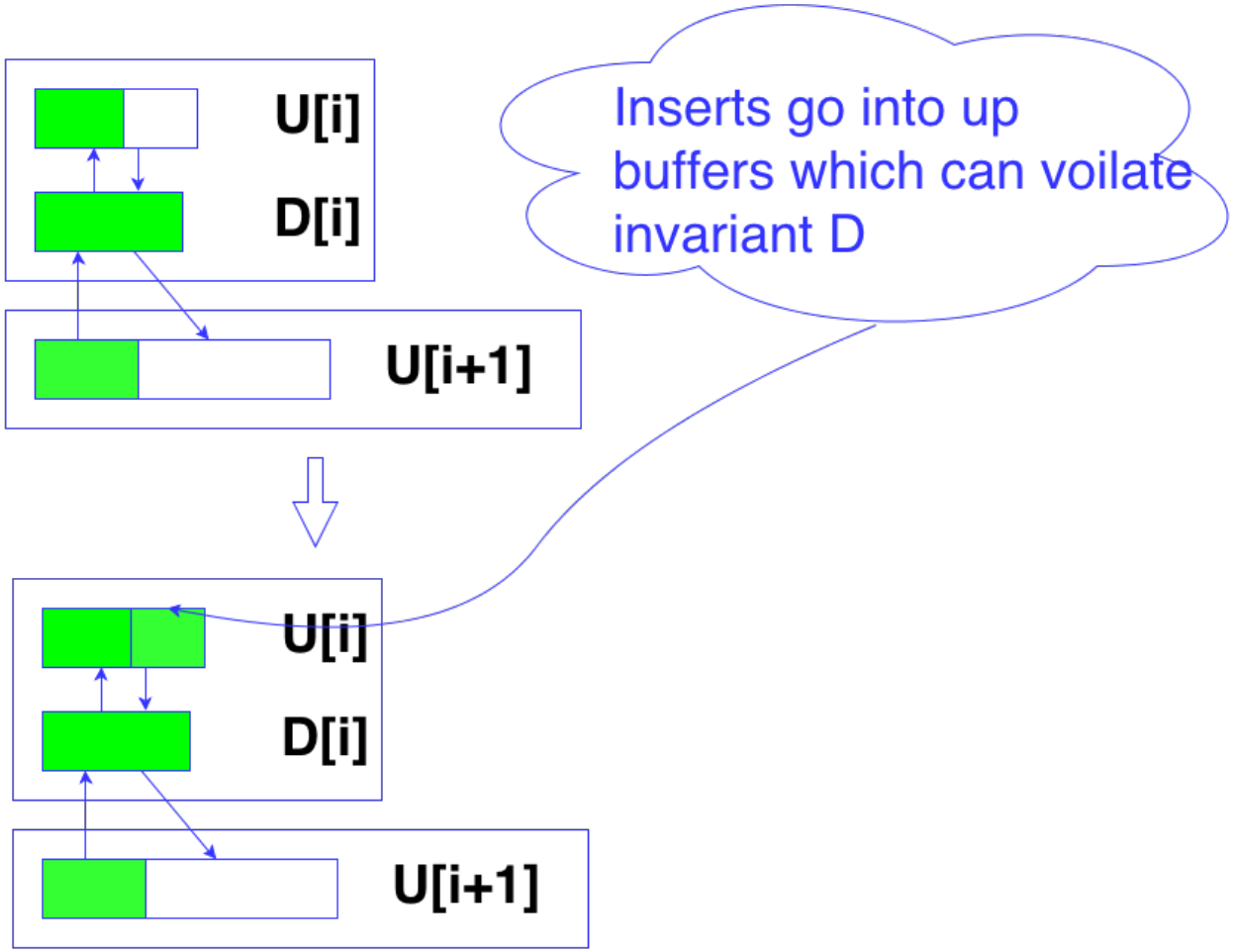
Invariant C - Down at least Half Filled



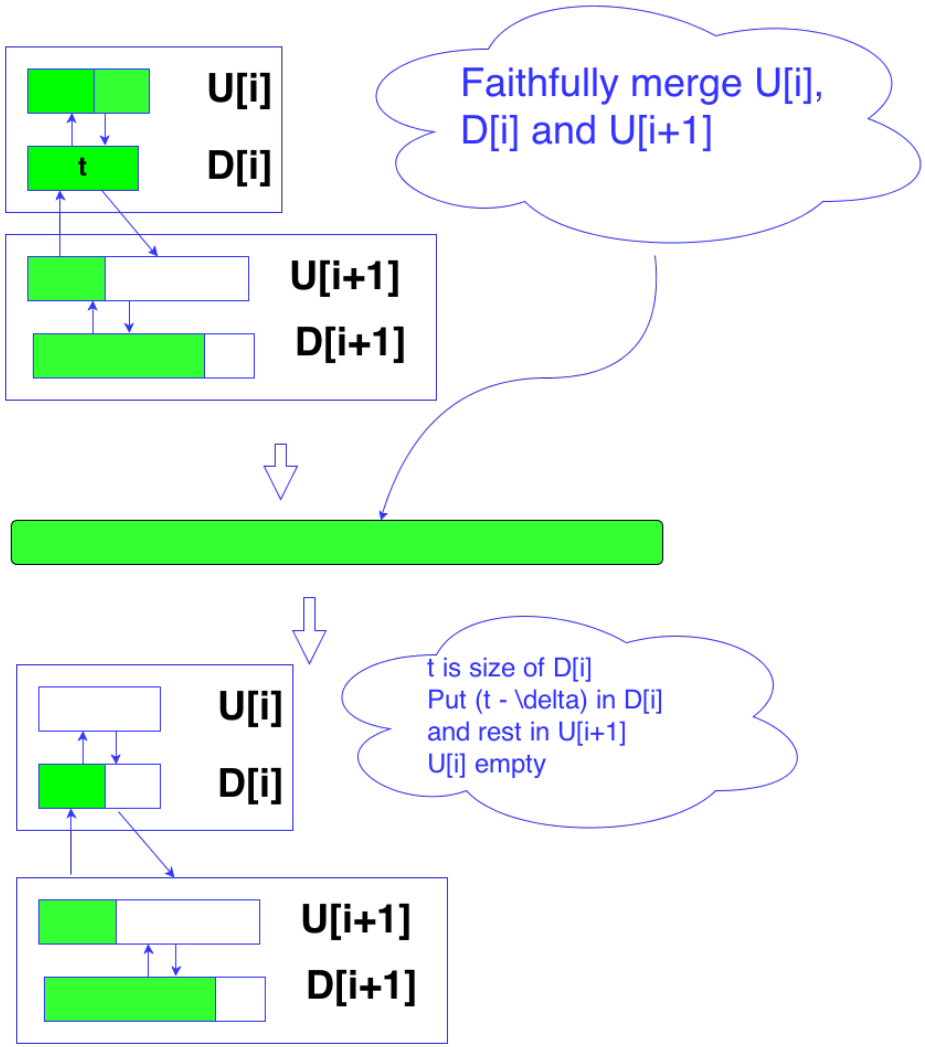
Invariant D - Up at most Half Filled



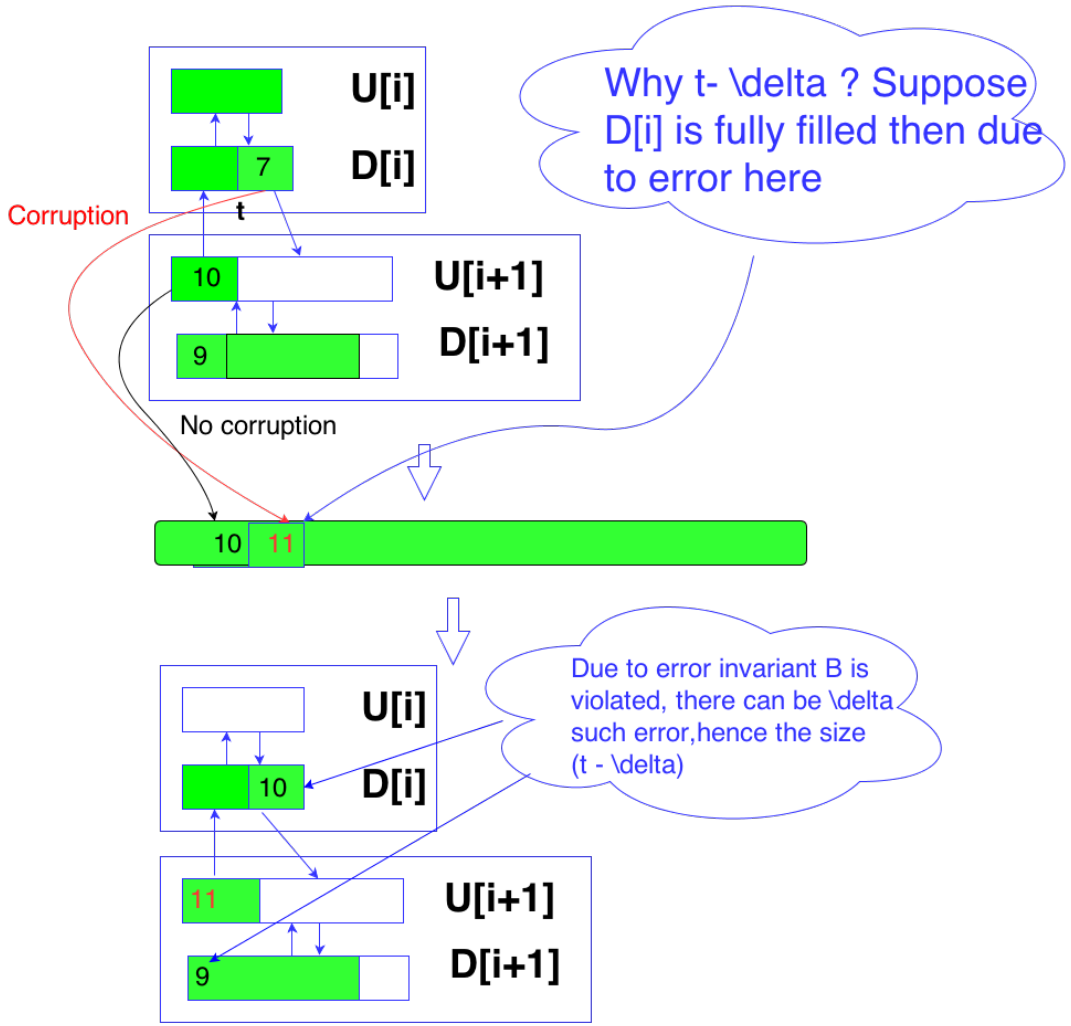
Insert


U[i]
D[i]
U[i+1]
Inserts go into up buffers which can voilate invariant D
U[i]
D[i]
U[i+1]

Insert


U[i]
D[i]
t
U[i+1]
D[i+1]
Faithfully merge U[i], D[i] and U[i+1]
t is size of D[i]
Put (t - \delta) in D[i]
and rest in U[i+1]
U[i] empty
U[i]
D[i]
U[i+1]
D[i+1]

Insert


U[i]
D[i]
7
t
Corruption
10
U[i+1]
D[i+1]
9
No corruption
10
11
Why t- \delta ? Suppose D[i] is fully filled then due to error here
U[i]
D[i]
10
11
U[i+1]
D[i+1]
9
Due to error invariant B is violated, there can be \delta such error,hence the size (t - \delta)

Push and *Pull*

- *Push* and *Pull* primitives used fro *Insert* and *DeleteMin*
- *Push* pushes elements to higher layers when Up at most half invariant is broken
- Similarly, *Pull* pulls elements from higher layer when Down at least half invariant is broken

Algorithm for *Push*

1: **function** PUSH(U_i)
2: $d_i = |D_i|$
3: $M = U_i \cup D_i \cup U_{i+1}$
4: $D_i = M[0..(d_i - \delta)]$
5: $U_i = []$
6: **if** $i = k$ **then**
7: $U_{k+1} = []$
8: $D_{k+1} = M[(d_i - \delta)..]$
9: k = k+1
10: **else**
11: $U_{i+1} = M[(d_i - \delta)..]$
12: **if** $U_{i+1} > s_{i+1}/2$ **then**
13: Push(U_{i+1})
14: **if** $D_i < s_i/2$ **then**
15: Pull(D_i)

Algorithm for *Pull*

1: **function** PULL(D_i)
2: $d_i = |D_i|$
3: $d_{i+1} = |D_{i+1}|$
4: $M = U_i \cup D_i \cup U_{i+1}$
5: $D_i = M[0..s_i]$
6: $D_{i+1} = [s_i..(d_{i+1} + d_i - \delta)]$
7: $U_{i+1} = [d_{i+1} + d_i - \delta..]$
8: **if** $U_{i+1} > s_{i+1}/2$ **then**
9: Push(U_{i+1})
10: **if** $D_{i+1} < s_{i+1}/2$ **then**
11: Pull(D_{i+1})

Insert

- For insertion we maintain a buffer I of size $(\delta + logn + 1)$ and simply append the new element to buffer
- When I is full we faithfully sort it and faithfully merge with U_0
- Call *Push* on U_0 if at most invariant is broken
- For *DeleteMin* return the minimum of the first $\delta + 1$ elements of D_0 and all of I
- if D_0 underflows, call *Pull*

Complexity

- *Insert* and *DeleteMin* takes $O(logn + \delta)$ amortized time
- *Push* and *Pull* on a buffer is called at most once each request
- Intuitively, $\Omega(s_i)$ operations happen between any two call at any level L_i which gives the amortized bounds
- Lower bound is proved to be the same
- Uses $O(n + \delta)$ space

Relation with Cache Oblivious

- This structure was inspired by cache oblivious priority queue by Bender et al in 2002
- There are several data structures which are adpated from their cache oblivious versions
- Even though resilient model does not have memory hierarchy
- One reason could be that cache oblivious structure use chunks of array to gain by locality and employs less pointers
- So they become amenable to be adapted to resilient version because then the small number of pointers can be stored in reliable memory
- Note that there is a tree based cache oblivious priority queue too

References

- L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In Proc. of the 34th Annual ACM Symposium on Theory of Computing, pages 268-276, 2002.
- I. Finocchi, F. Grandoni, and G. F. Italiano. Optimal resilient sortingand searching in the presence of memory faults. In Proceedings of the 33rd International Colloquium on Automata, Languages and Programming, volume 4051 of Lecture Notes in Computer Science, pages 286-298. Springer, 2006
- A. G. Jorgensen. Data Structures: Sequence Problems, Range Queries and Fault Tolerance. PhD Dissertation, Aarhus University, Department of Computer Science, Denmatk, pages 79-87, 2010.