A Simple and Practical Concurrent Non-blocking Unbounded Graph with Linearizable Reachability Queries

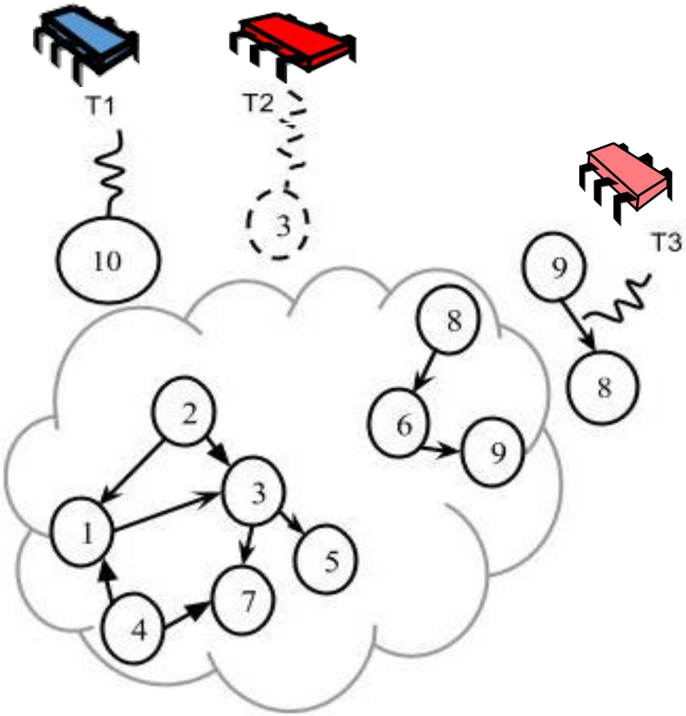
Bapi Chatterjee[*], Sathya Peri[†], **Muktikanta Sa**[†], Nandini Singhal[‡]
[*]Institute of Science and Technology Austria, bapi.chatterjee@ist.ac.at
[†]Dept. of CS&E, IIT Hyderabad, India, {sathya_p, cs15resch11012}@iith.ac.in
[‡]Microsoft (R&D) Pvt. Ltd., Bangalore, India, nandini12396@gmail.com

Graphs are Everywhere...

Common real world objects can be modeled as graphs, which build the pairwise relations between objects.

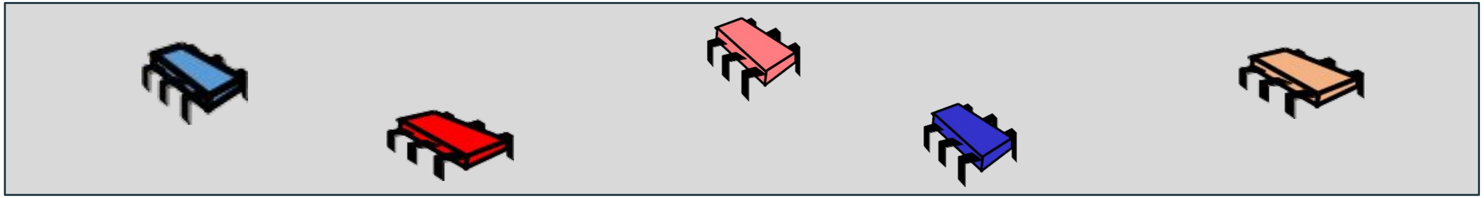
Graph algorithms applied in many applications, including social networks, communication networks, VLSI design, graphics, etc.

Often these graphs are dynamic in nature and the updates are real-time.



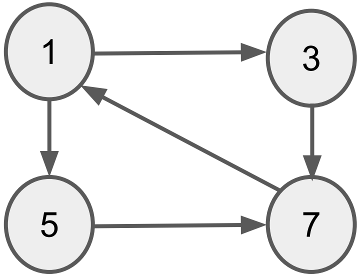
The System Model

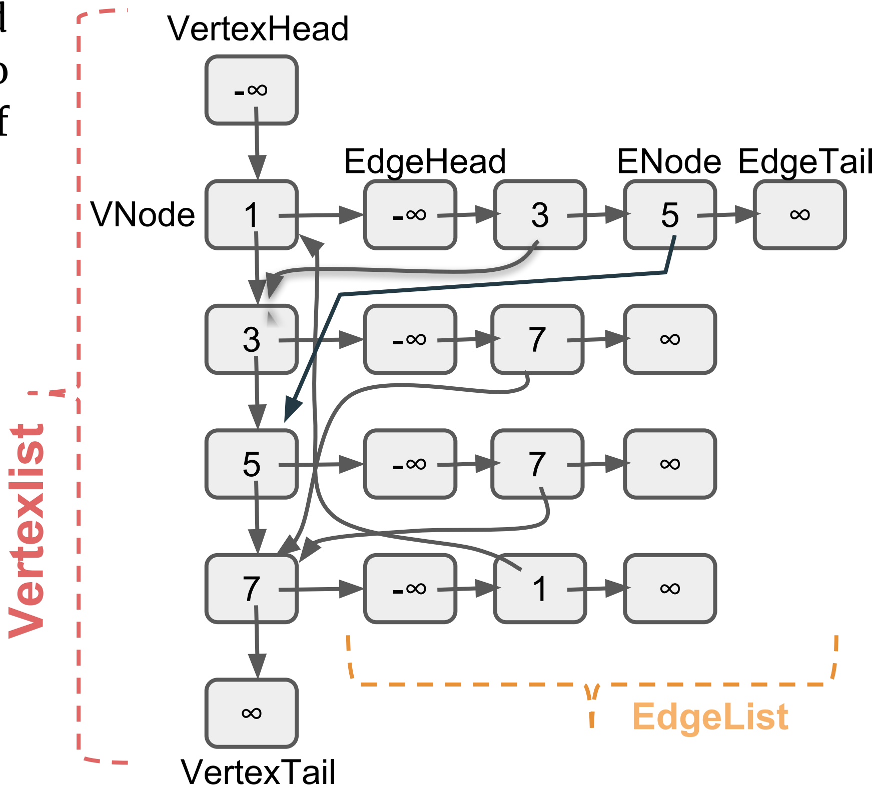
- ➔ Asynchronous shared-memory model with a finite set of *p* processors accessed by a finite set of *n* threads.
- ➔ The non-faulty threads communicate with each other by invoking methods on the shared objects.
- ➔ Execution on a shared-memory multi-processor system which supports atomic *read*, *write*, *fetch-and-add* (FAA) and *compare-and-swap* (CAS) instructions..



The Data Structure

A directed graph $G = (V, E)$ represented by its adjacency list which enables it to grow (up to the availability of memory) and sink at the runtime.





The ADT Operations

1. **AddVertex (k)**
2. **RemoveVertex (k)**
3. **AddEdge (k, l)**
4. **RemoveEdge (k**

Update Operations

Challenge

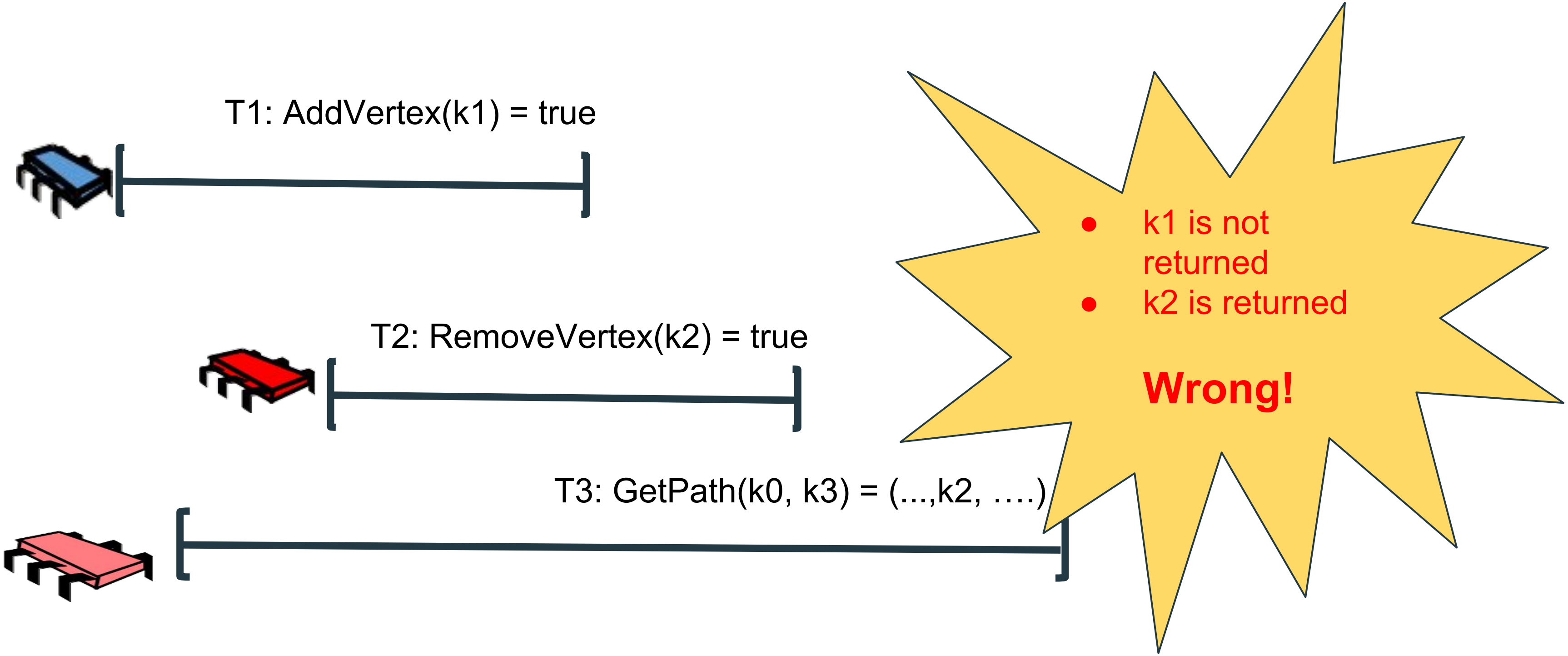
Consistency of a traversal with concurrent Updates in the Graph

5. **ContainsVertex**
6. **ContainsEdge (k**
7. **GetPath(k, l)**

Graph Traversal

non-update Operations

Consistency Example



T1: AddVertex(k1) = true

T2: RemoveVertex(k2) = true

T3: GetPath(k0, k3) = (...,k2, ….)

- k1 is not returned
- k2 is returned

Wrong!

Correctness

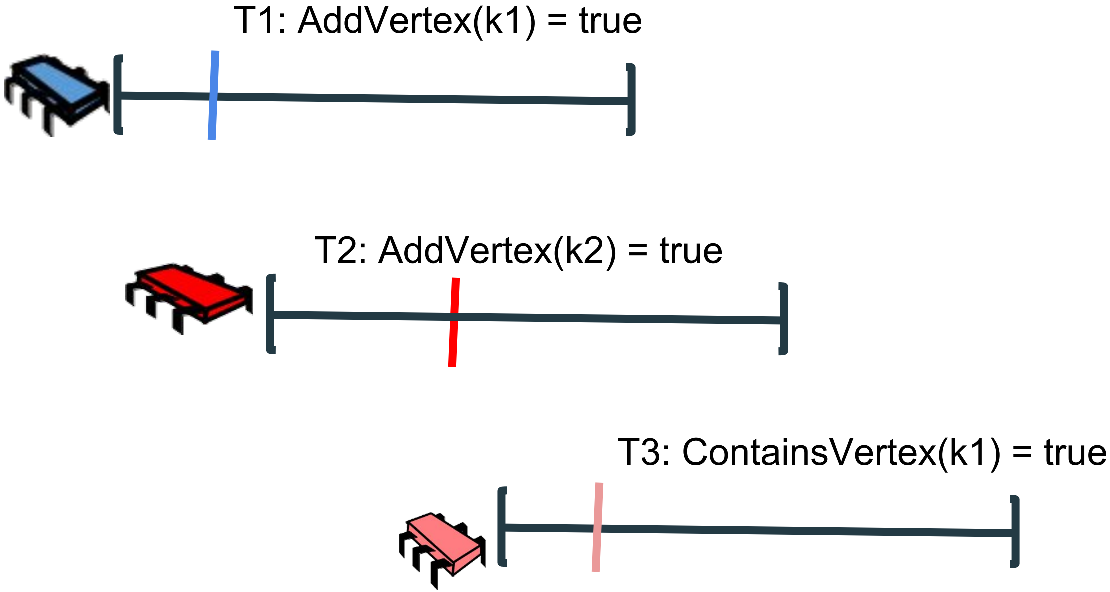
- ➔ The ADT operations implemented by the data structure are represented by their *invocation* and *return* steps.

- ➔ For an arbitrary concurrent execution of a set of ADT operations should satisfy the consistency framework **linearizability.**

- ➔ Assign an atomic step as a *linearization point* (LP) inside the execution interval of each of the operations and show that the data structure invariants are maintained across the LPs.

- ➔ An arbitrary concurrent execution is equivalent to a valid sequential execution obtained by ordering the operations by their LPs.

Linearizability Example



T1: AddVertex(k1) = true

T2: AddVertex(k2) = true

T3: ContainsVertex(k1) = true

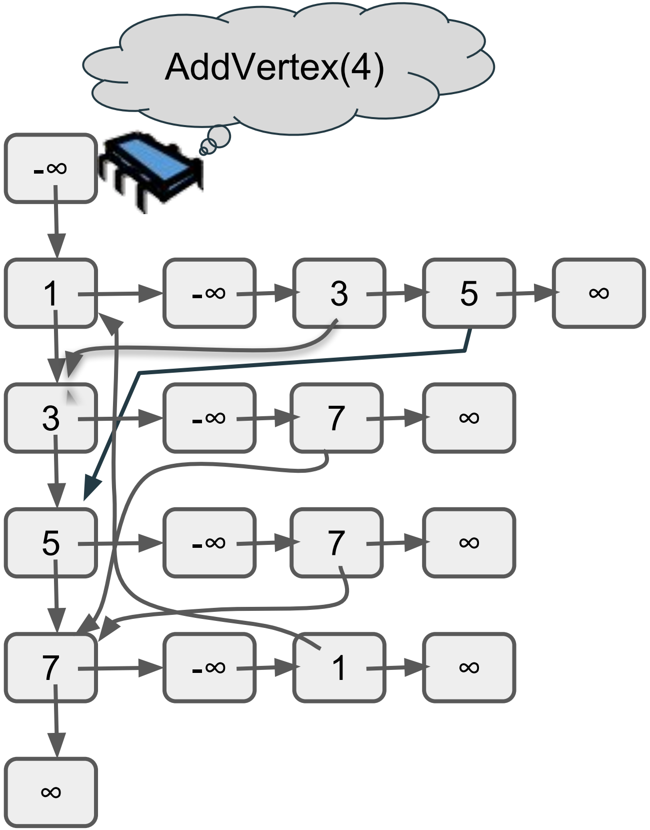
Progress Guarantee

A method is **wait-free** if it guarantees that every call finishes its execution in a finite number of steps.

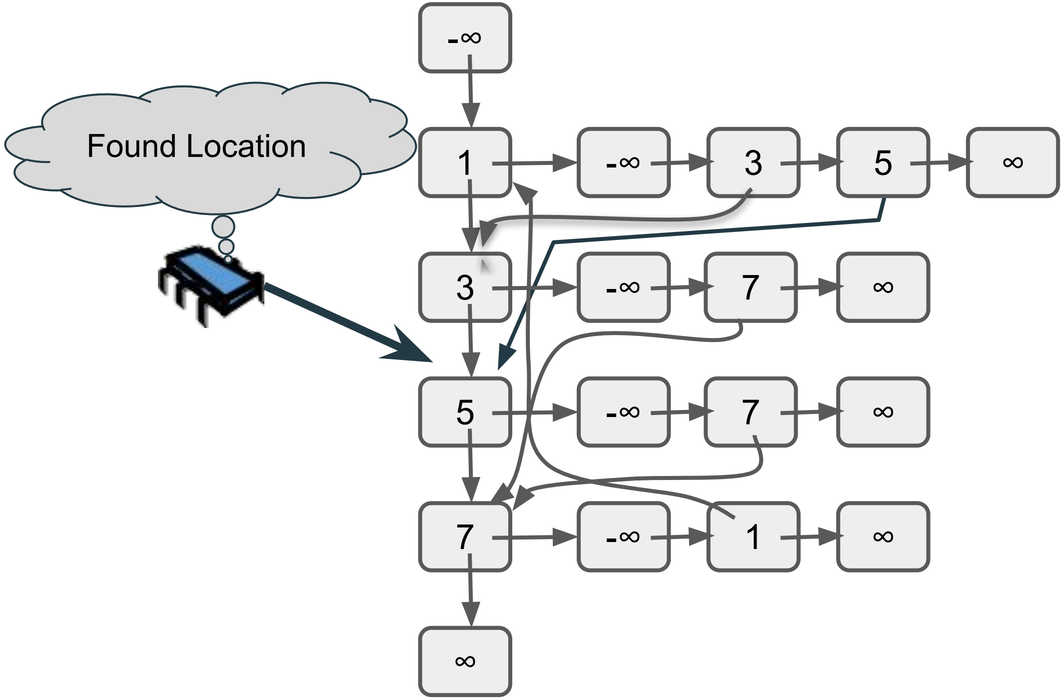
A method is **lock-free** if it guarantees that infinitely often some method call finishes in a finite number of steps.

A method is **obstruction-free** if, from any point after which it executes in isolation, it finishes in a finite number of steps (method call executes in isolation if no other threads take steps).

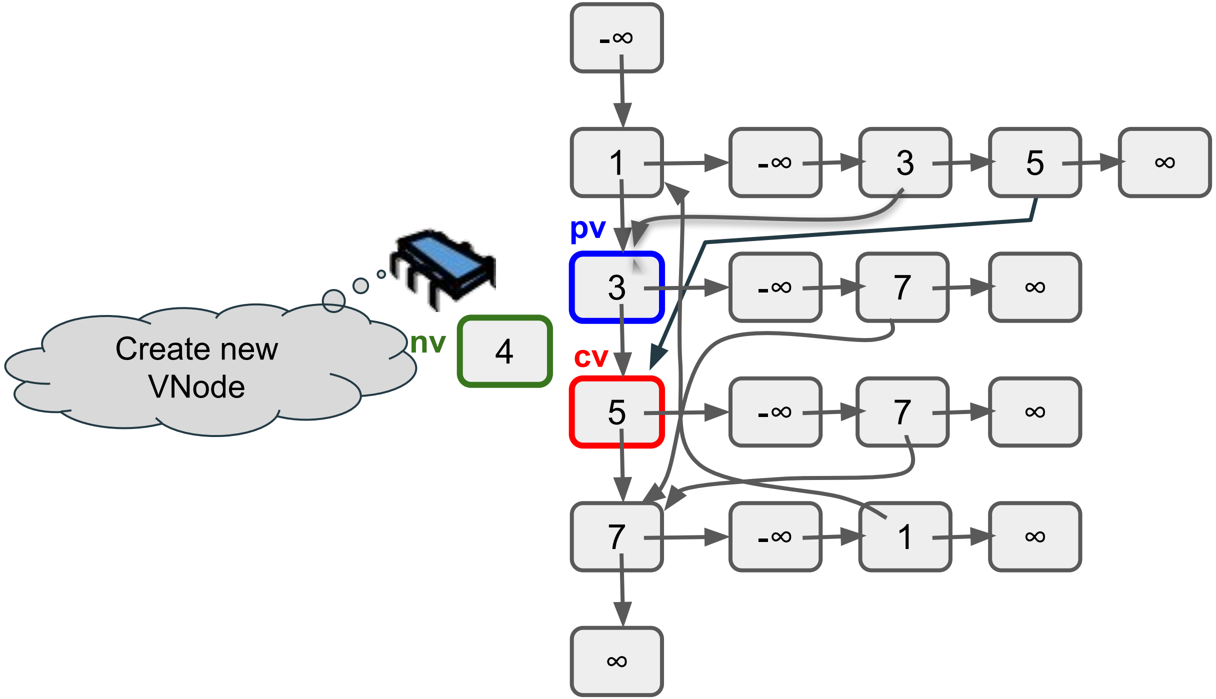
Add Vertex



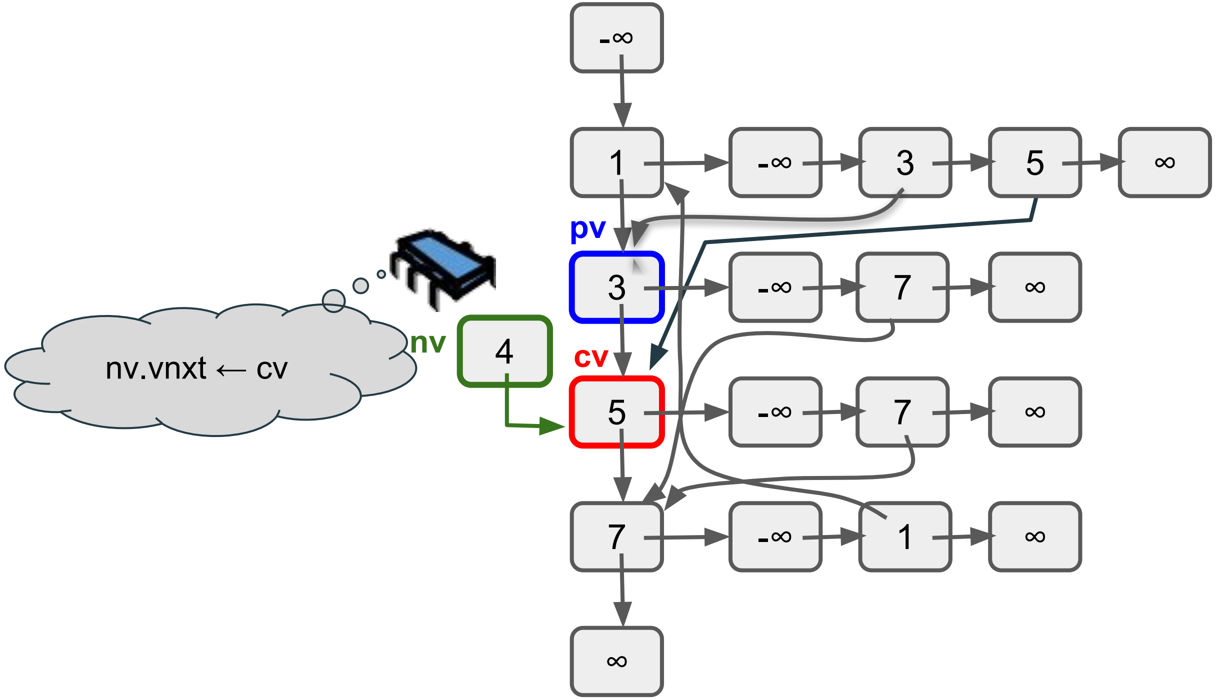
Add Vertex



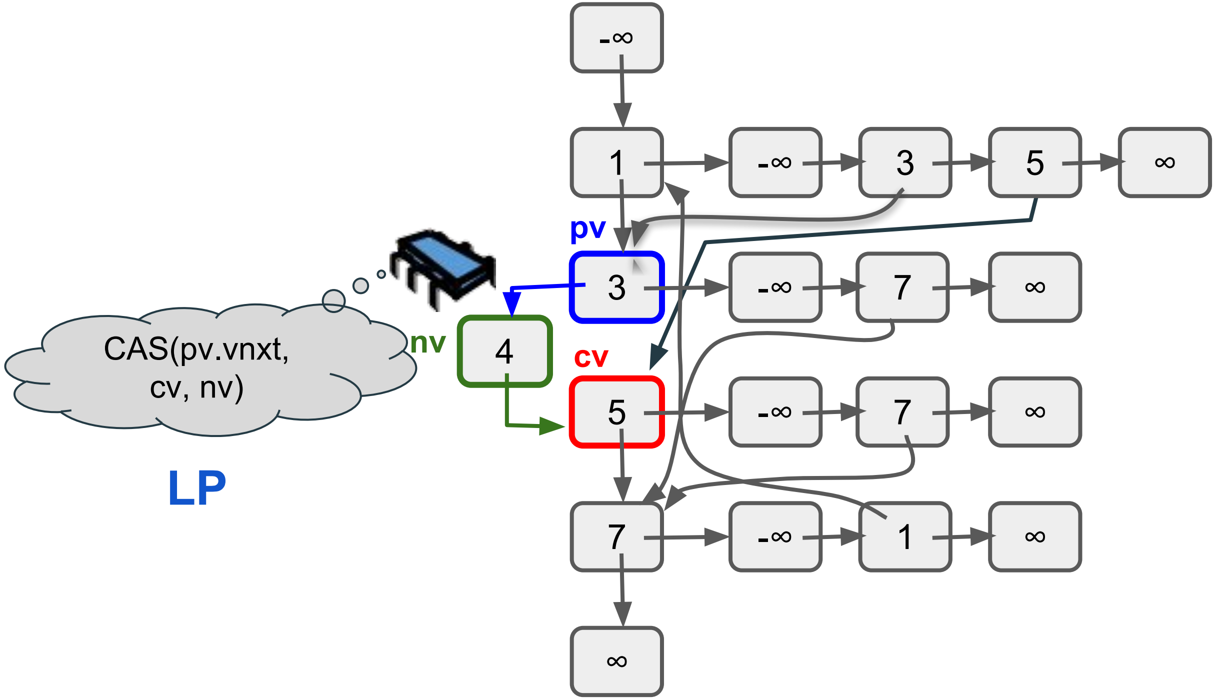
Add Vertex



Add Vertex

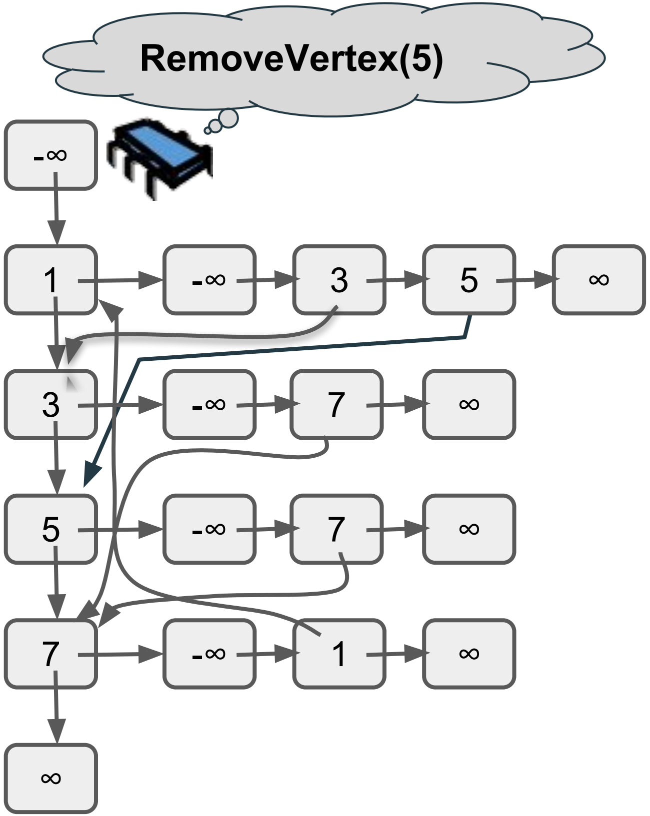


Add Vertex

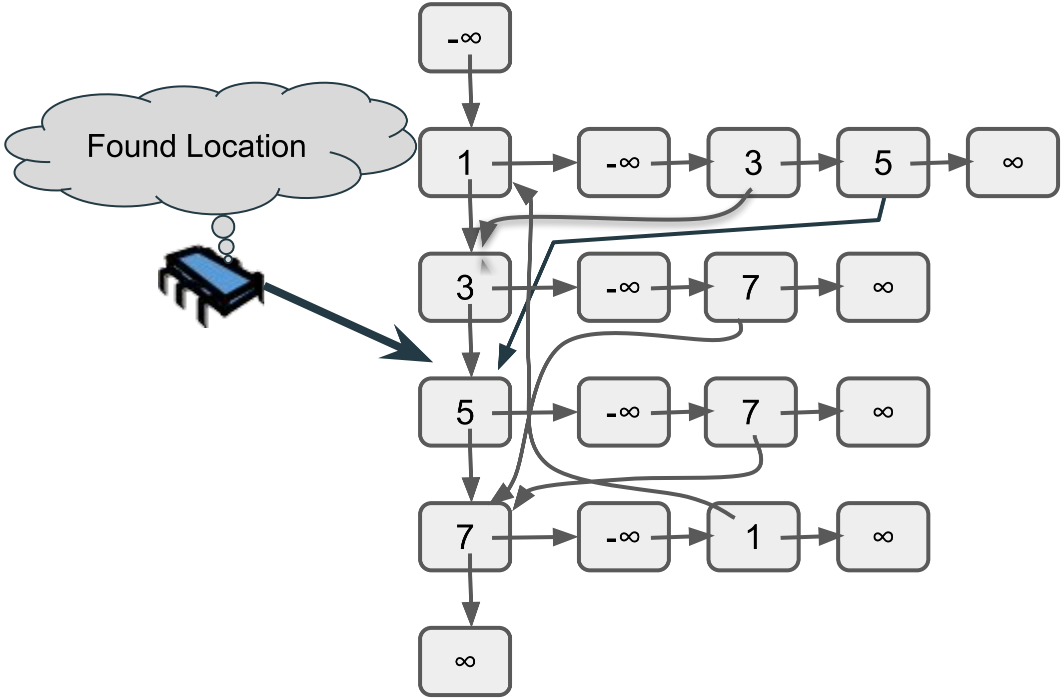


LP

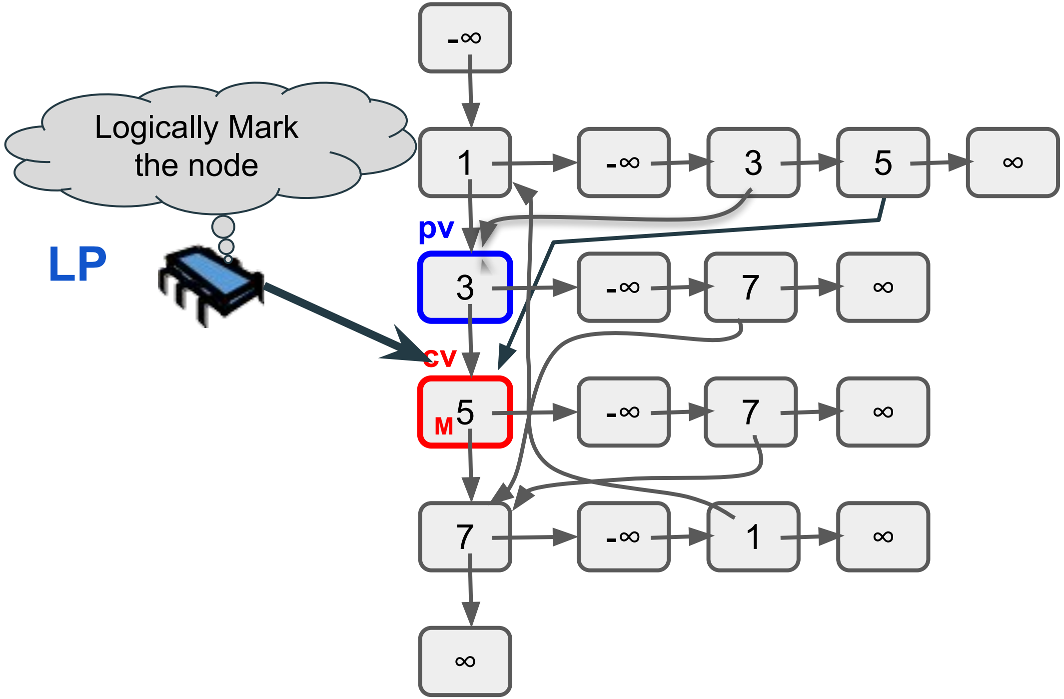
Remove Vertex



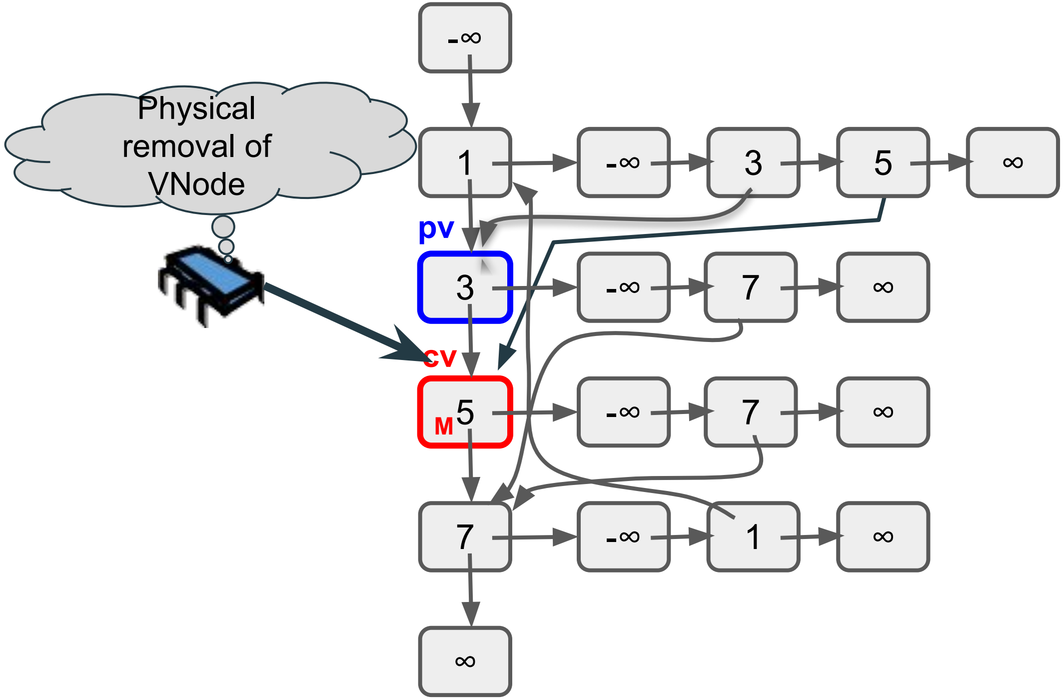
Remove Vertex



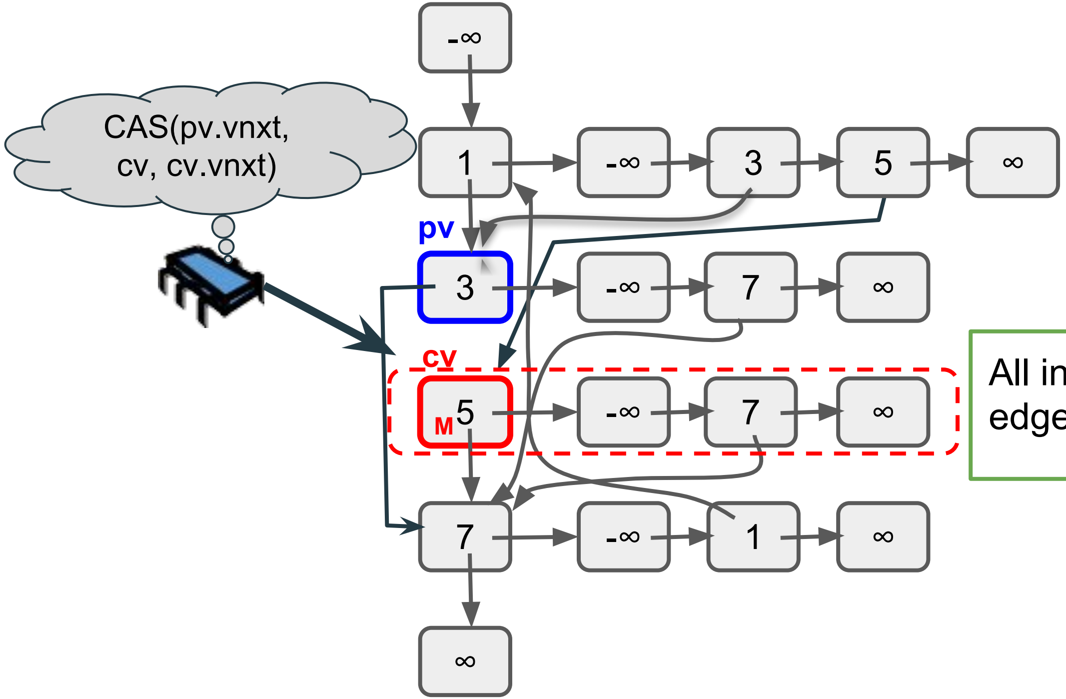
Remove Vertex



Remove Vertex

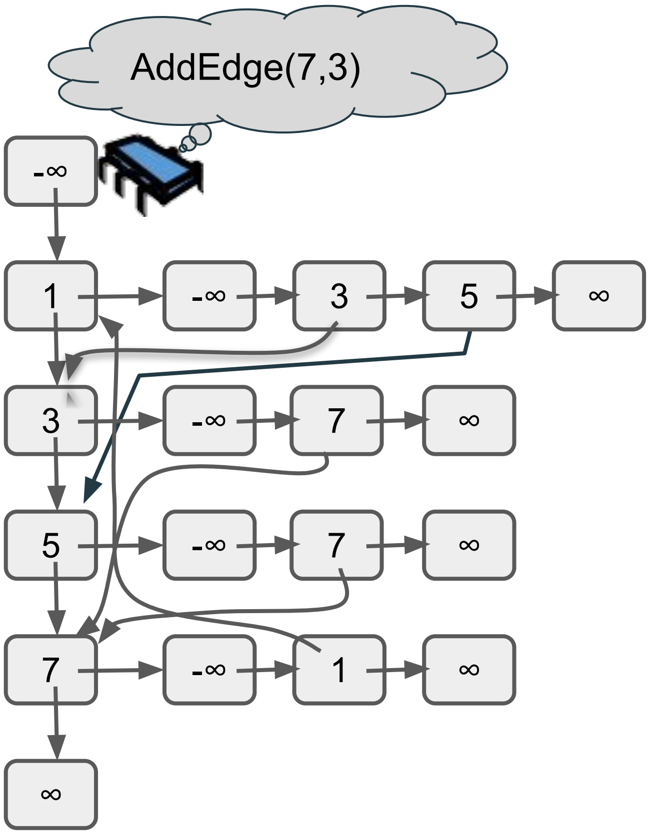


Remove Vertex



All incoming and outgoing edges are logically removed.

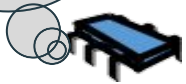
Edge Operations

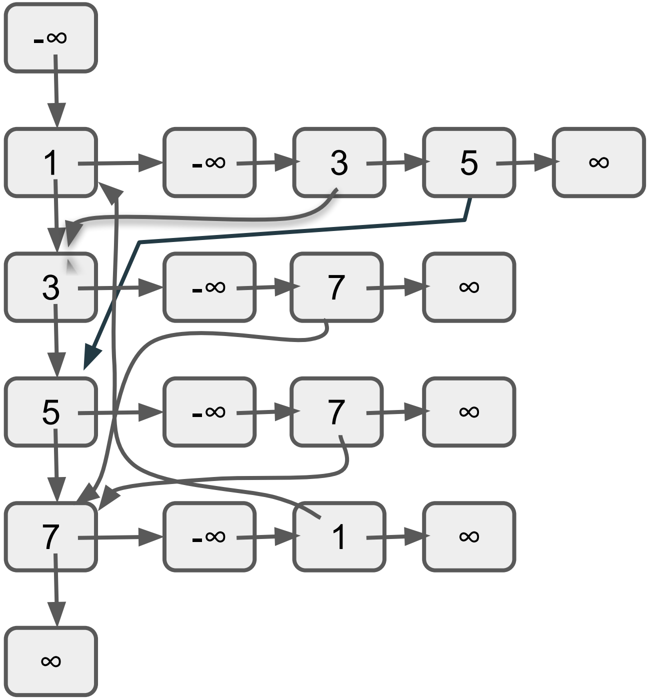


Add Edge

AddEdge(7,3)

Find and verify the presence of vertices v(7) and v(3) in the vertex-list.





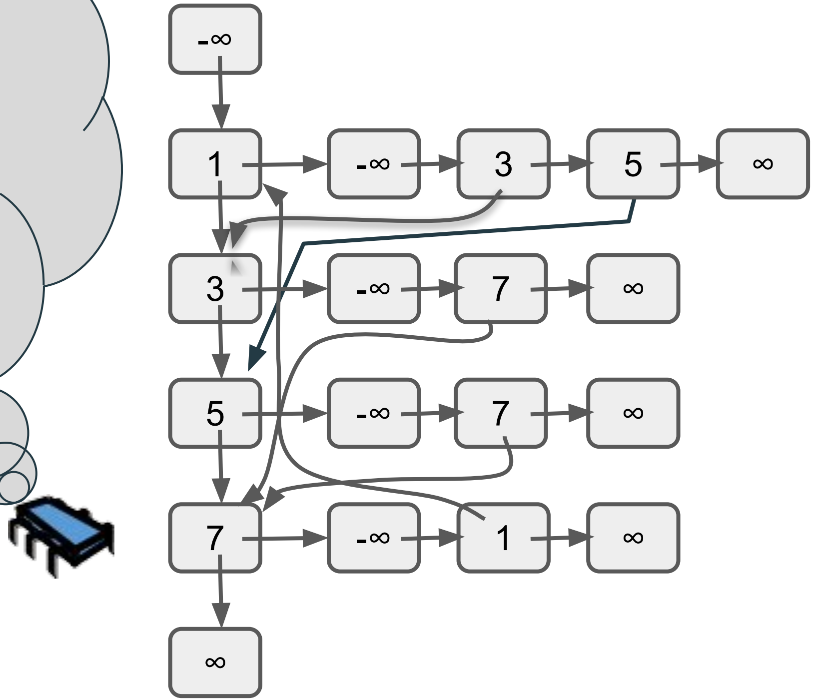
Add Edge

AddEdge(7,3)

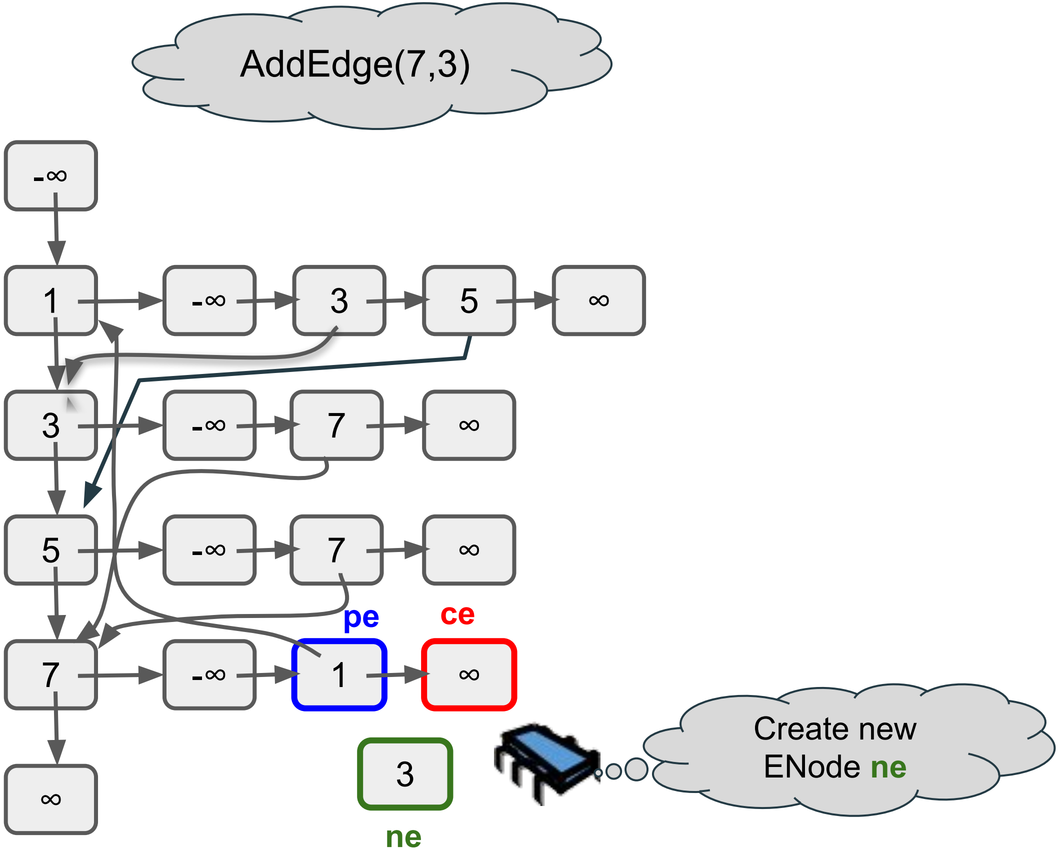
traverse down an edge-list of v(7) using **locE** and physically removes two kind of logically removed ENodes:

(a). logically removed VNode.

(b) logically removed ENodes,

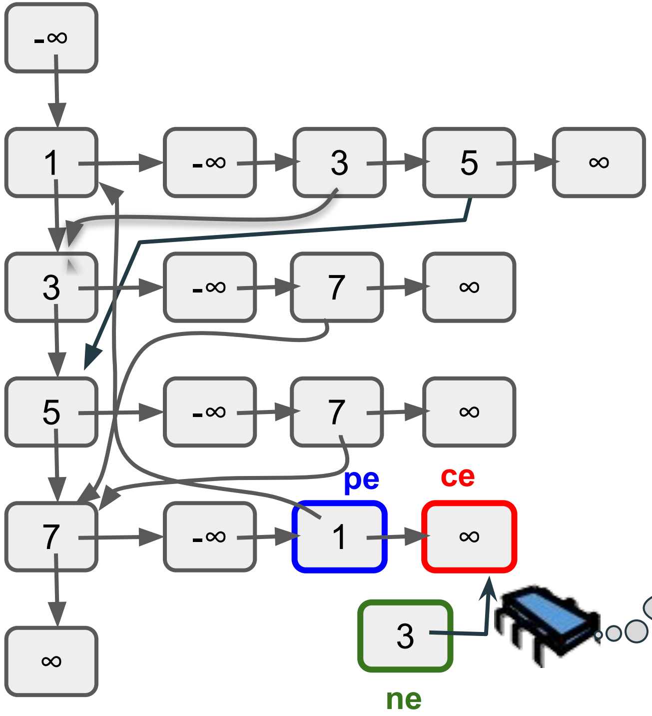


Add Edge



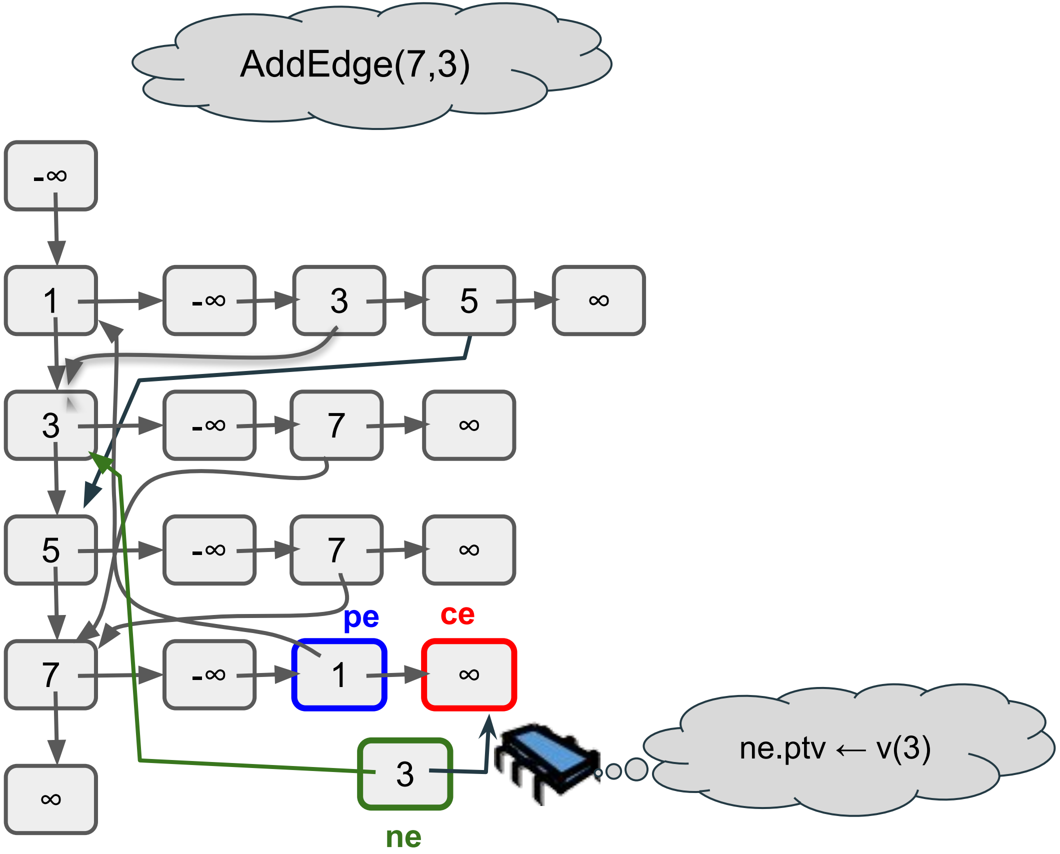
Add Edge

AddEdge(7,3)

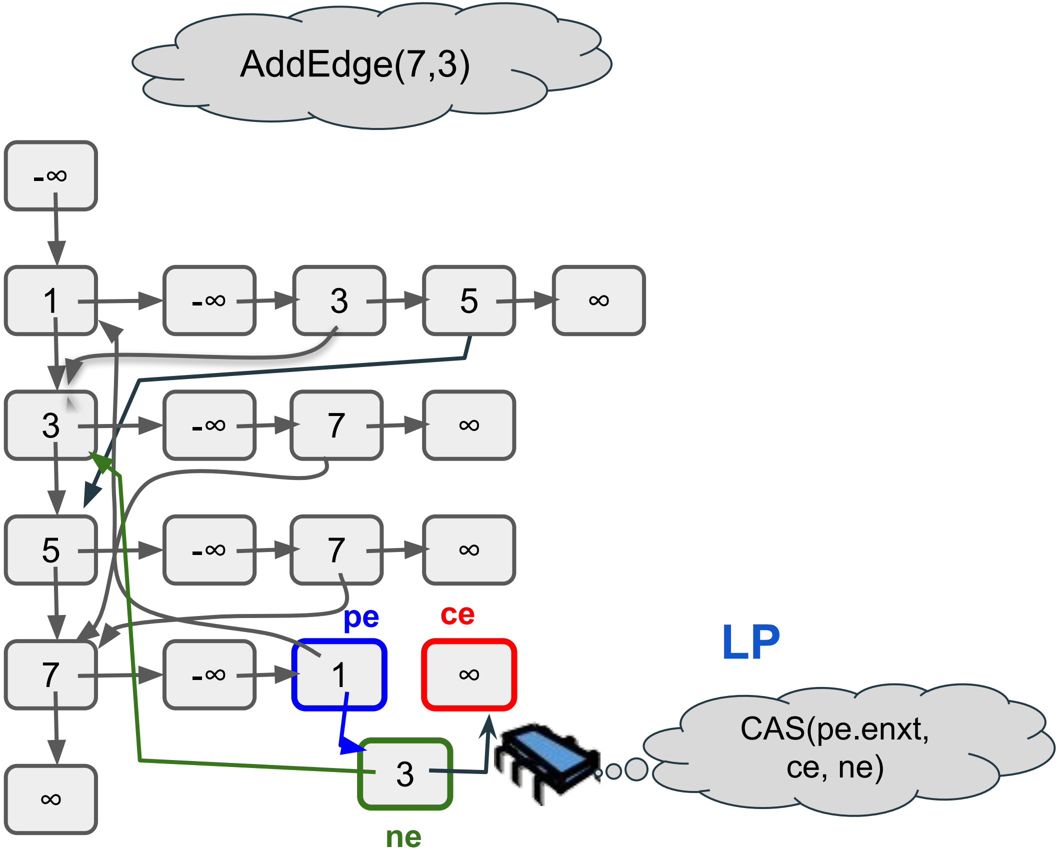


ne.enxt ← ce

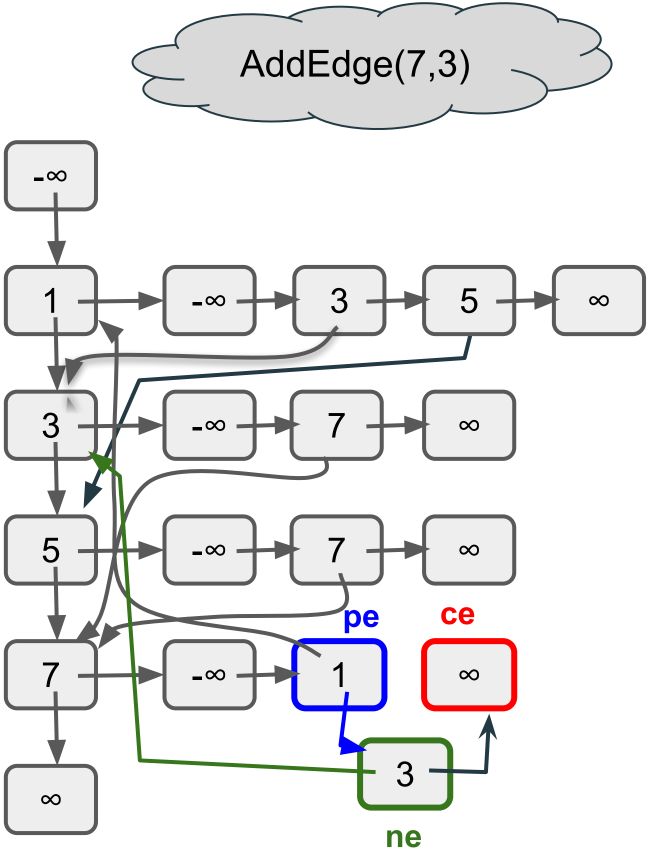
Add Edge



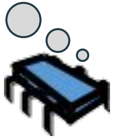
Add Edge



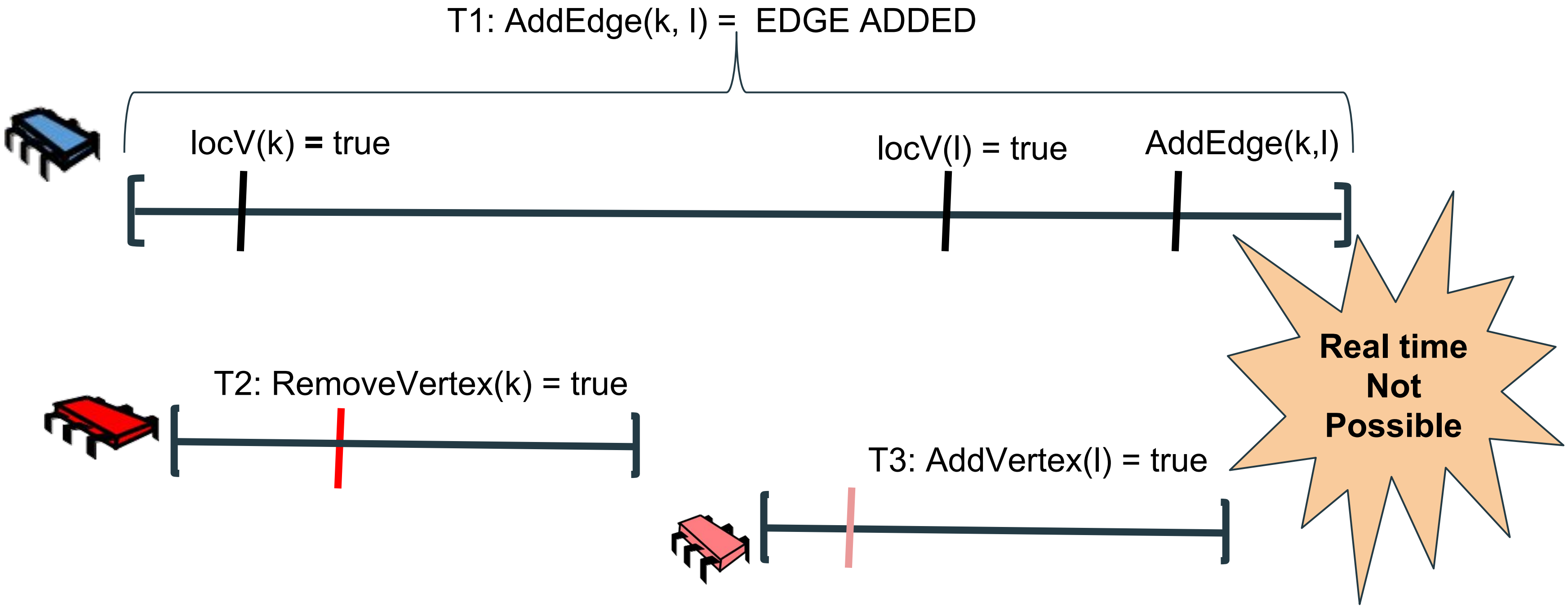
Add Edge



Increment a **counter** at v(7)



AddEdge Conflicts with Vertex Modifications



T1: AddEdge(k, l) = EDGE ADDED

locV(k) = true

locV(l) = true

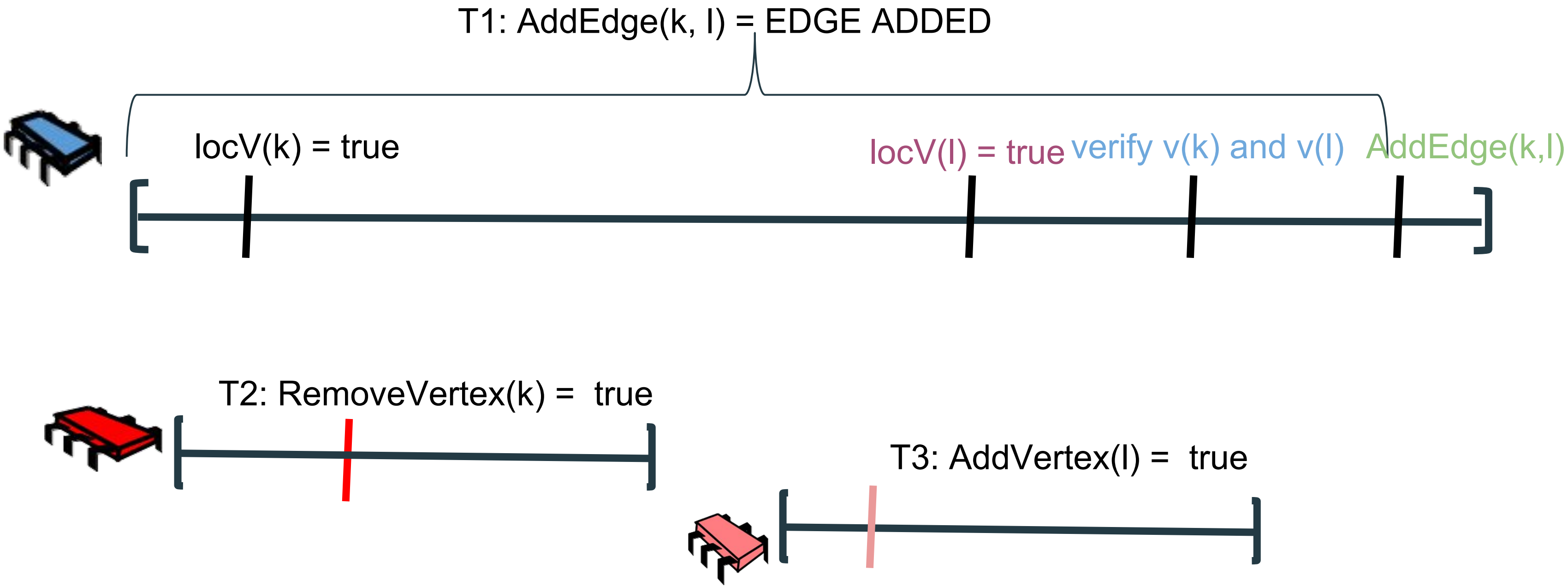
AddEdge(k,l)

T2: RemoveVertex(k) = true

T3: AddVertex(l) = true

Real time Not Possible

Synchronization of AddEdge with Vertex oparations



T1: AddEdge(k, l) = EDGE ADDED

locV(k) = true

locV(l) = true

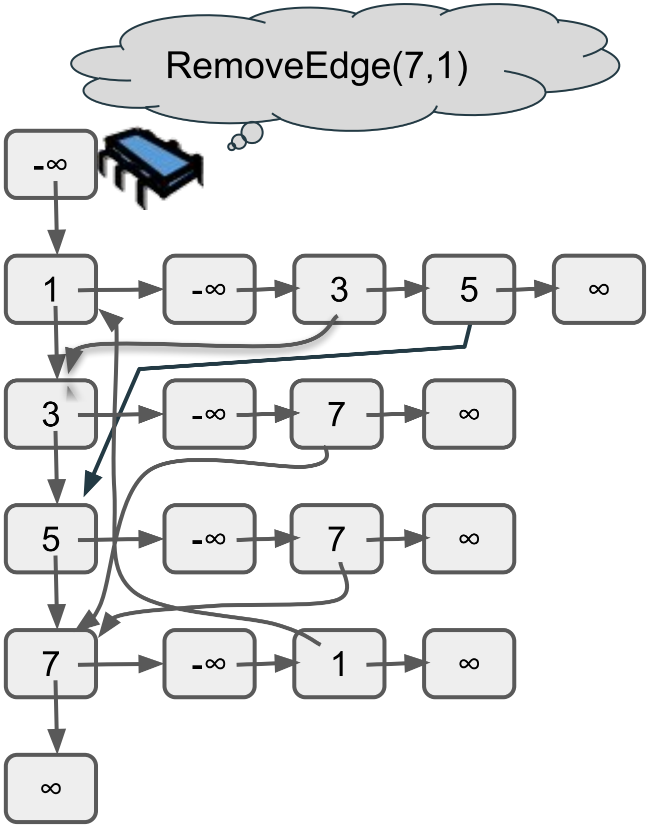
verify v(k) and v(l)

AddEdge(k,l)

T2: RemoveVertex(k) = true

T3: AddVertex(l) = true

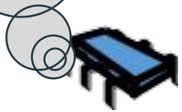
Remove Edge

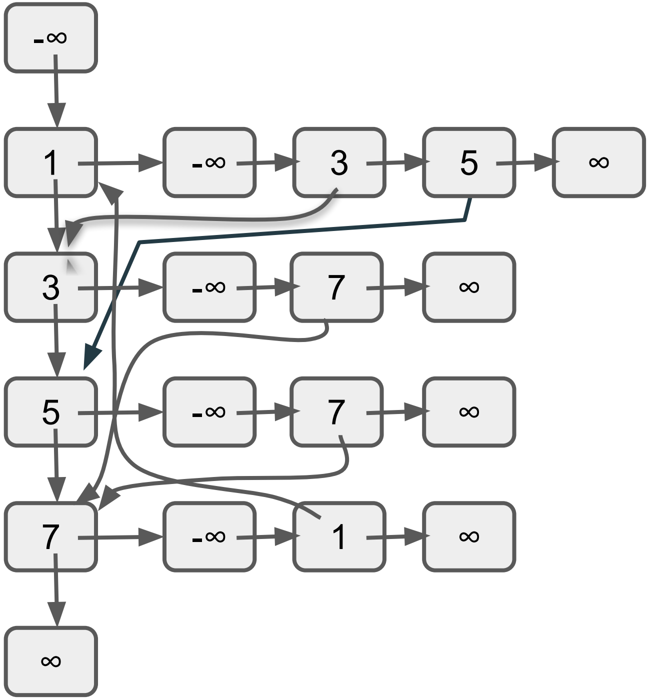


Remove Edge

RemoveEdge(7,1)

verify the presence of vertices v(7) and v(1) in the vertex-list using **ConVPlus**().





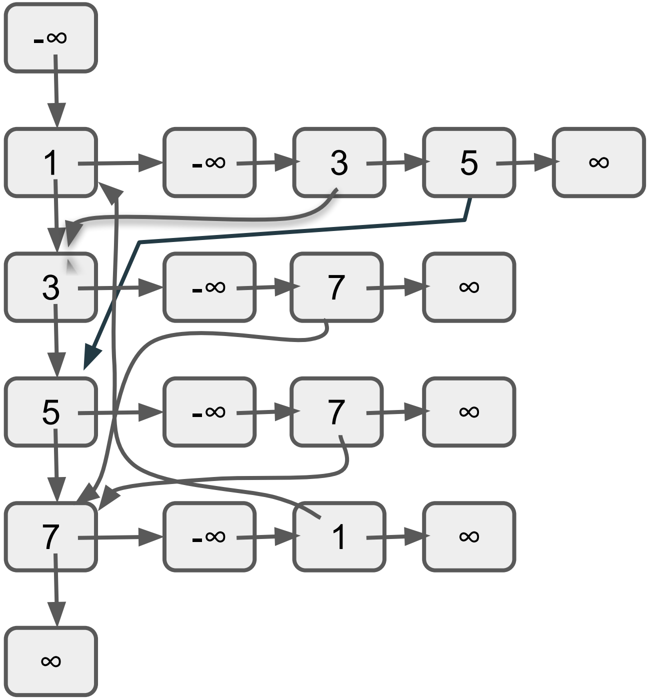
Remove Edge

traverse down an edge-list of v(7) using **locE** and physically removes two kind of logically removed ENodes:

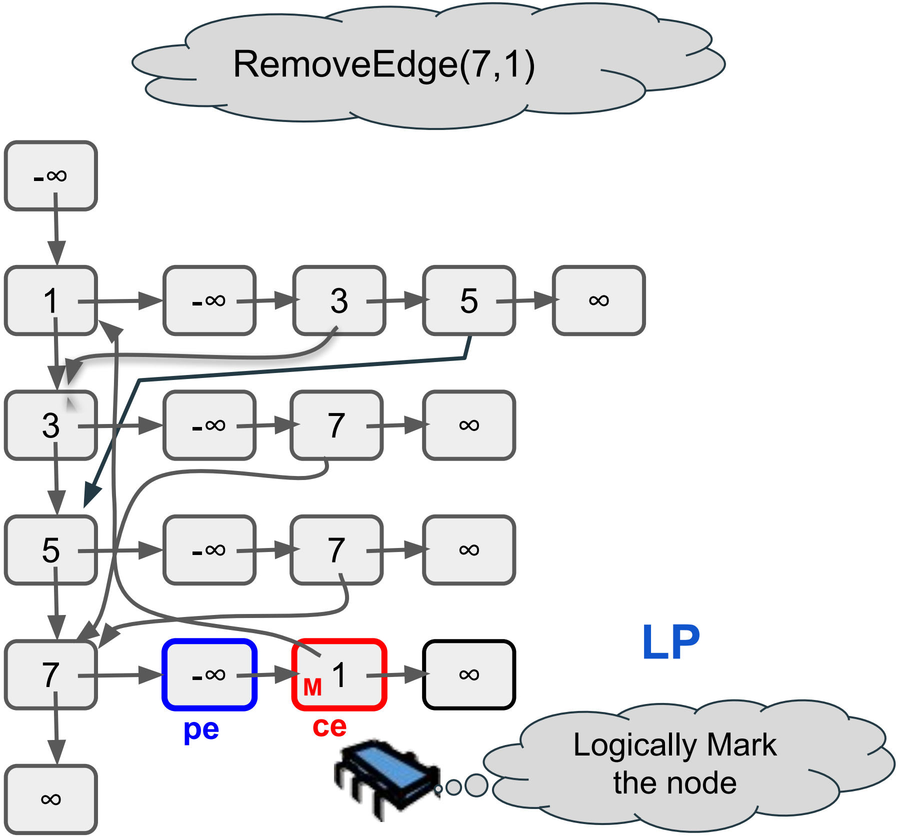
(a). logically removed VNode.

(b) logically removed ENodes,

RemoveEdge(7,1)

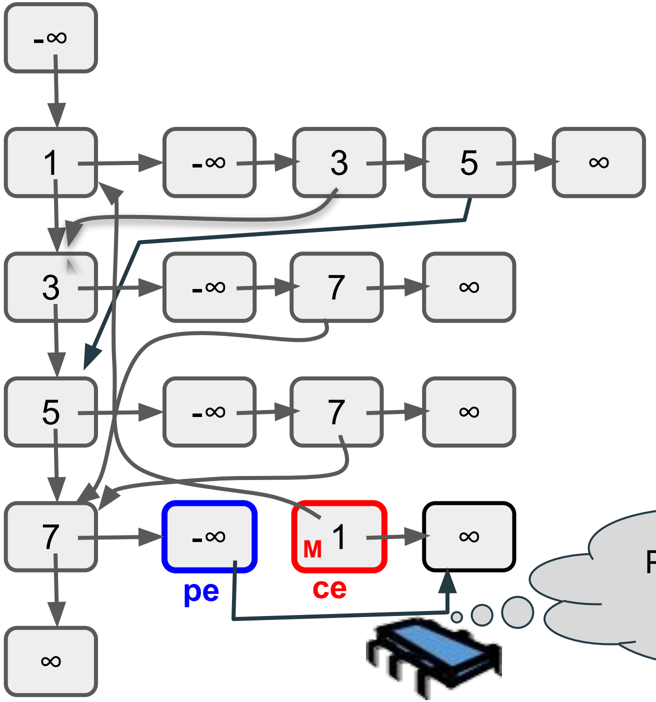


Remove Edge



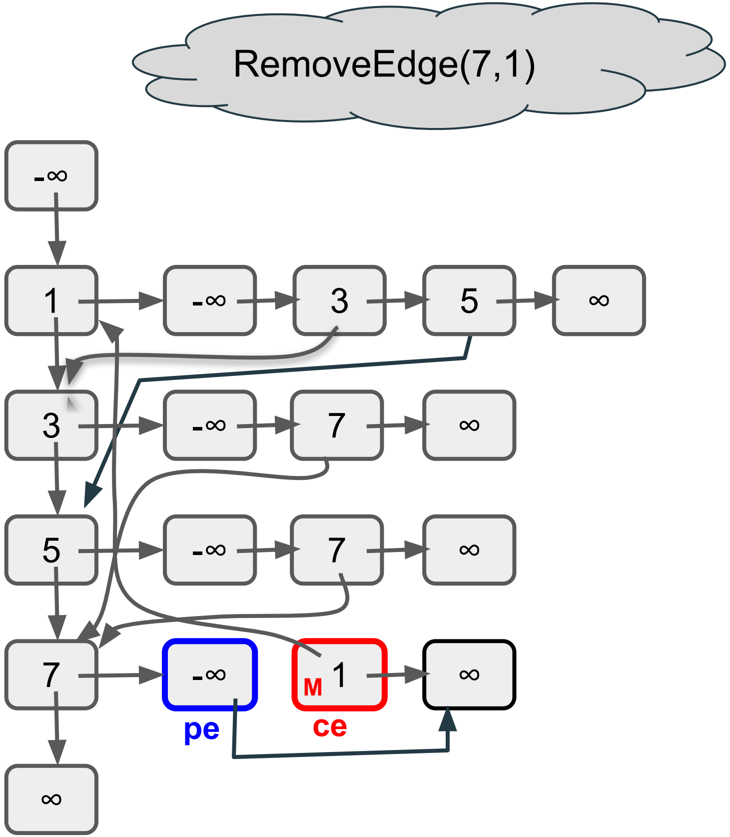
Remove Edge

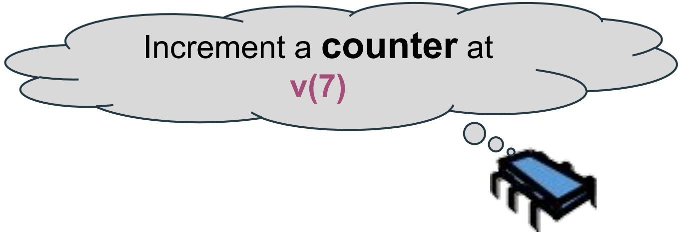
RemoveEdge(7,1)



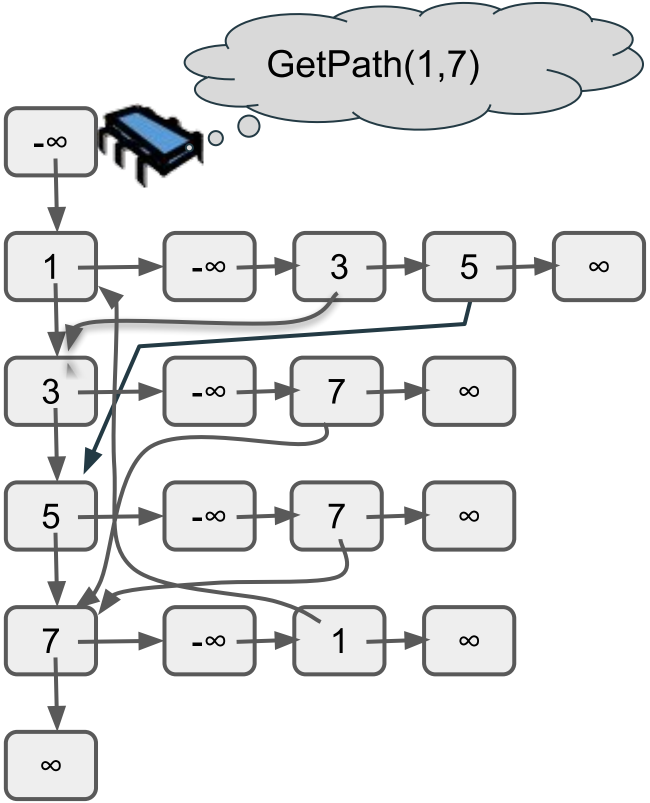
Physical removal of ENode
CAS(pe.enxt, ce, ce.enxt)

Remove Edge





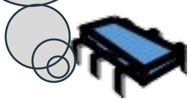
Reachability Query

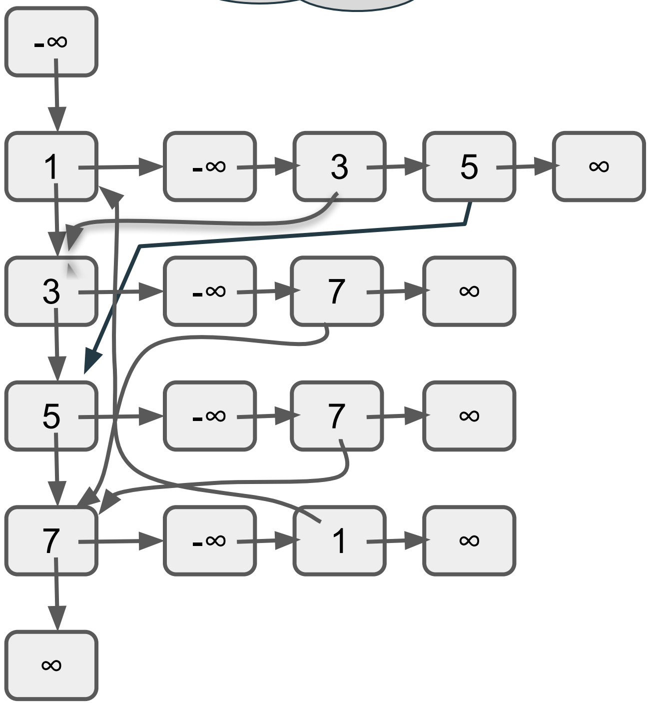


Reachability Query

GetPath(1,7)

verify the presence of vertices v(1) and v(7) in the vertex-list using **ConVPlus**().

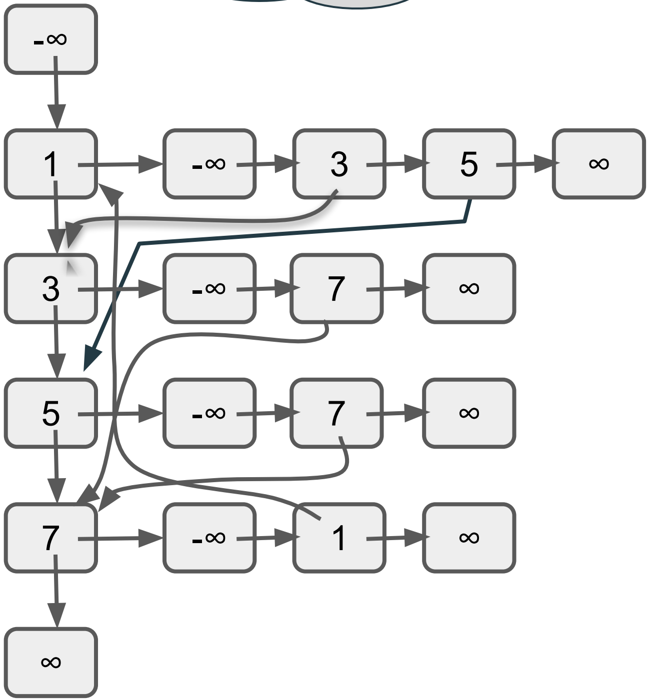


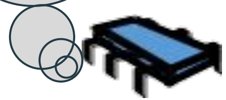


Reachability Query

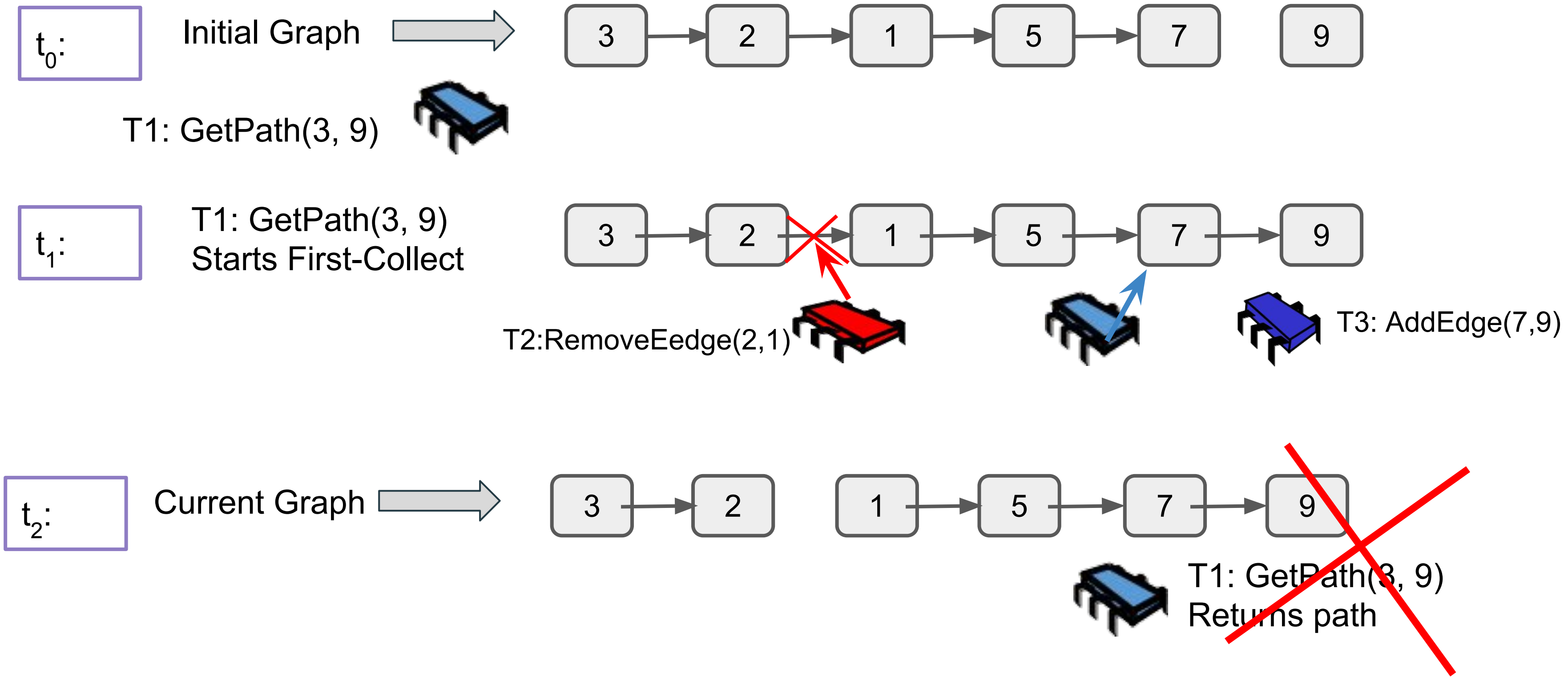
GetPath(1,7)

Collect the Path(Single-Collect) using traversal algorithm: **BFS**





Problem with Single Collect



t0: Initial Graph

T1: GetPath(3, 9)

t1: T1: GetPath(3, 9)
Starts First-Collect

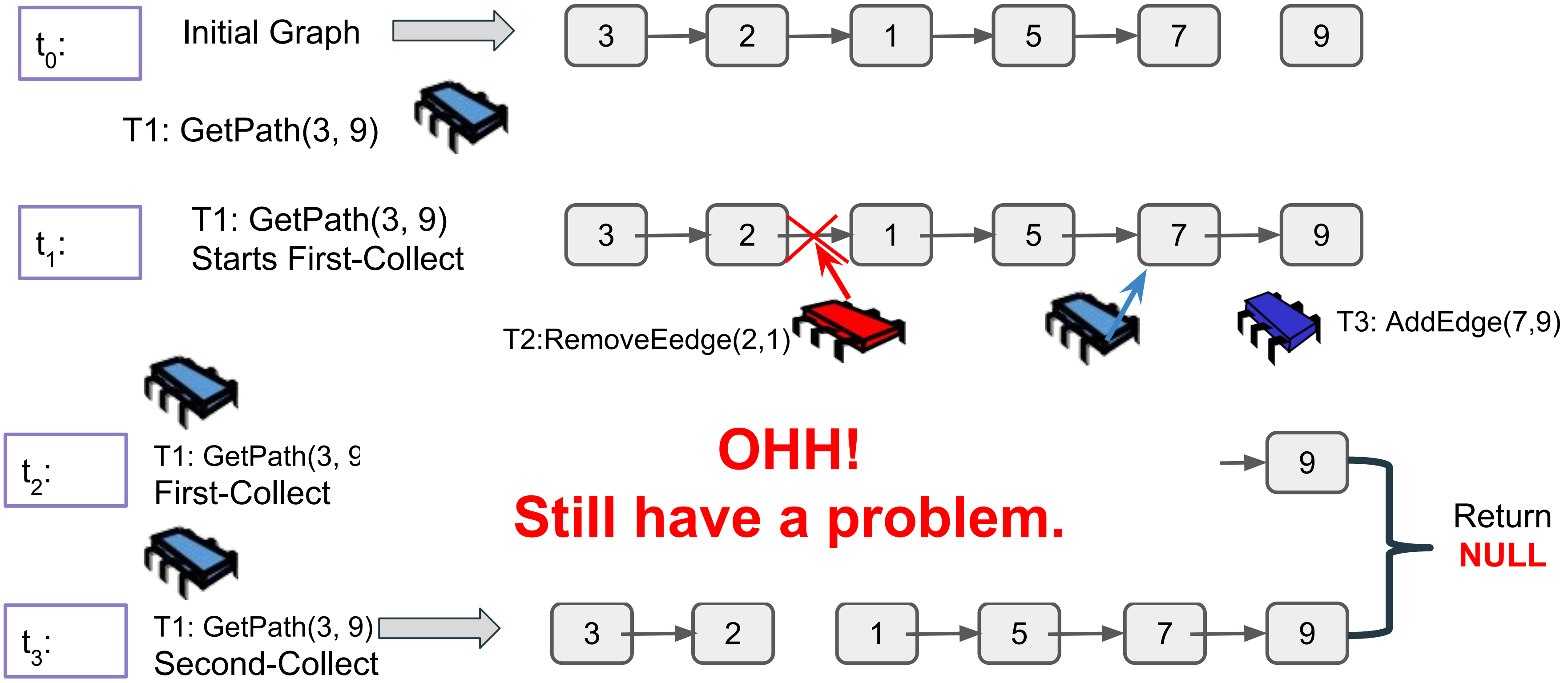
T2:RemoveEedge(2,1)

T3: AddEdge(7,9)

t2: Current Graph

T1: GetPath(3, 9)
Returns path

Need Double Collect



t_0:

Initial Graph

T1: GetPath(3, 9)

t_1:

T1: GetPath(3, 9)
Starts First-Collect

T2:RemoveEedge(2,1)

T3: AddEdge(7,9)

t_2:

T1: GetPath(3, 9
First-Collect

OHH!

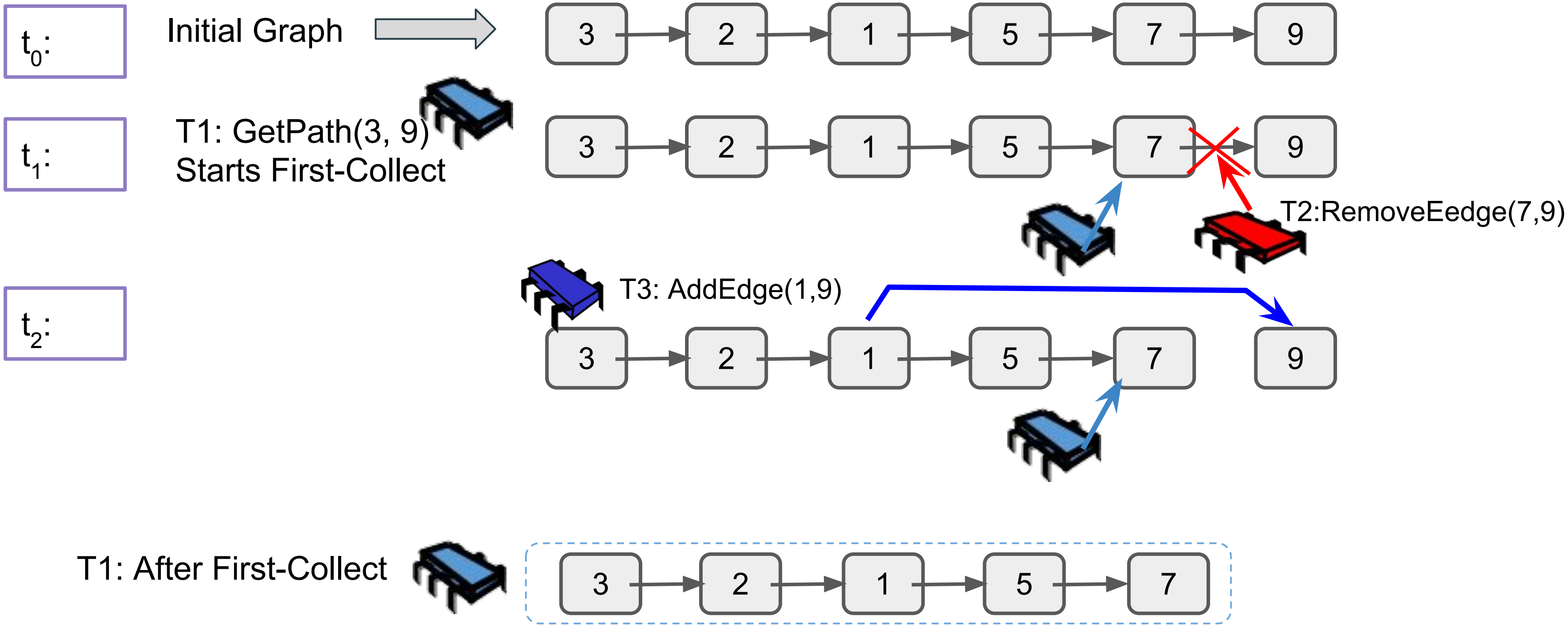
Still have a problem.

t_3:

T1: GetPath(3, 9)
Second-Collect

Return
NULL

Double Collect Problem



t_0:

Initial Graph

t_1:

T1: GetPath(3, 9)
Starts First-Collect

T2:RemoveEedge(7,9)

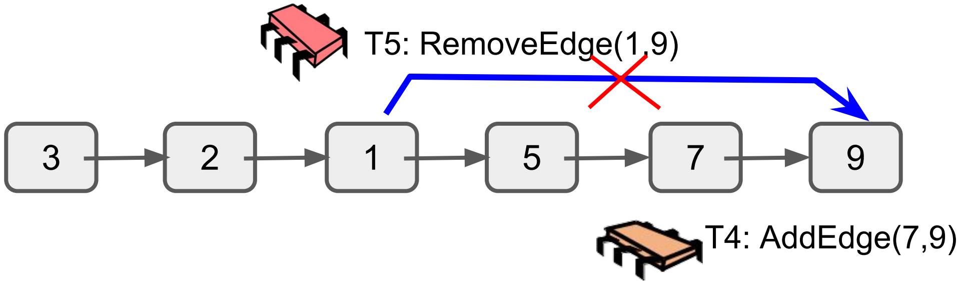
T3: AddEdge(1,9)

t_2:

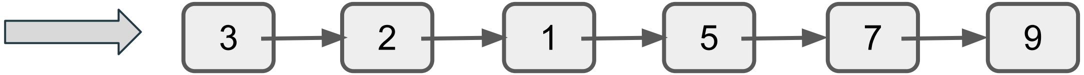
T1: After First-Collect

Problem ...

t_3:

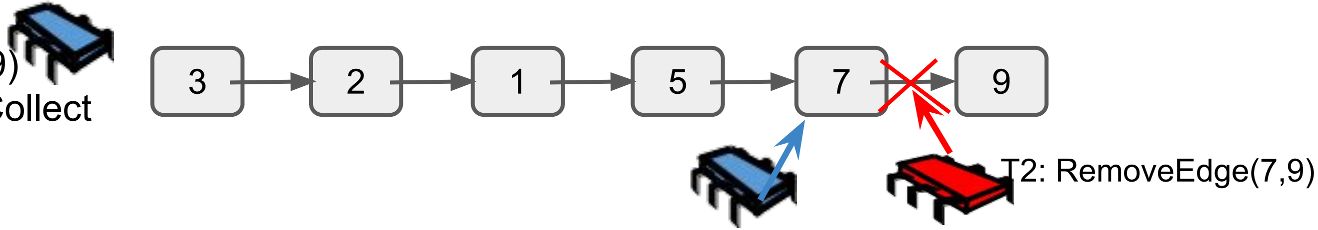


Graph has been restored



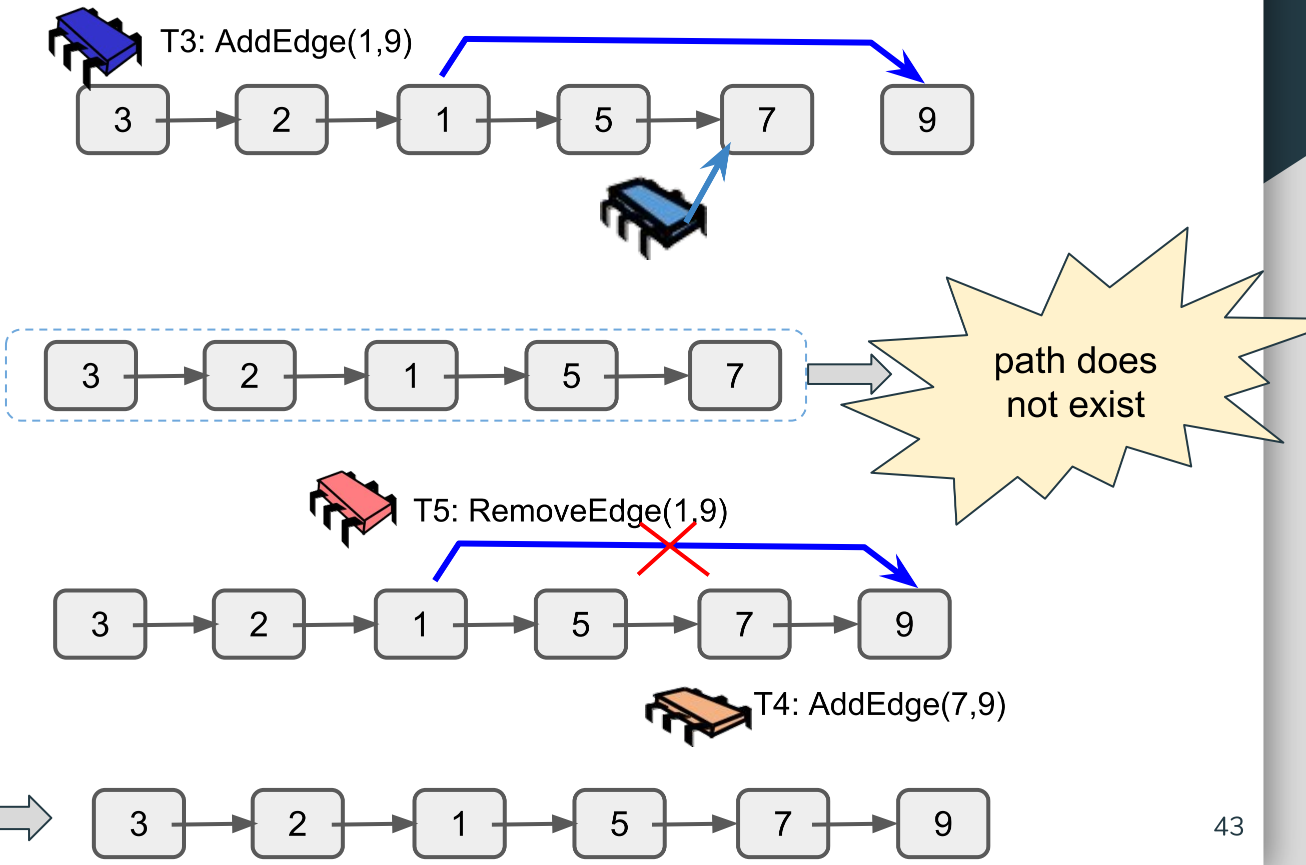
t_4:

T1: GetPath(3, 9)
Starts Second-Collect



Problem ...

t_5:



T1: After Second-Collect

t_6:

Graph has been restored

Solution

To solve these issues...

1. We take double collect.
2. In each **scan** we collect BFS-tree which is a partial snapshot.

3. To capture the modifications.
    3.1. We have a **counter** associated with each vertex.
    3.2. Whenever any edge operations happens the counter incremented.

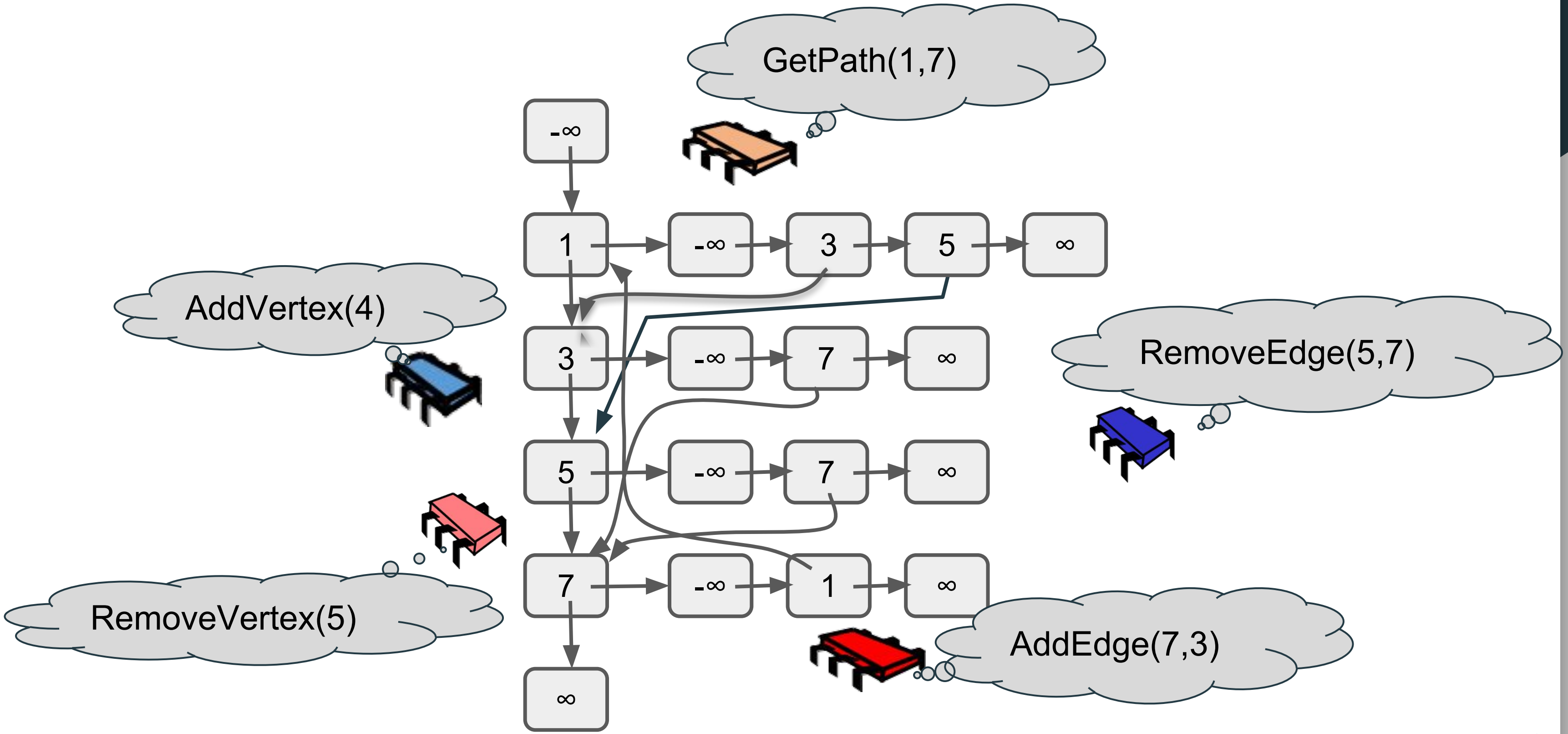
4. To verify the double collect we compare with BFS-tree alone with counter.

Solution ...

If the both the double collects are same

5.1. We have valid snapshot
5.2. We analyse the the valid snapshot for the presence or absence of the path.

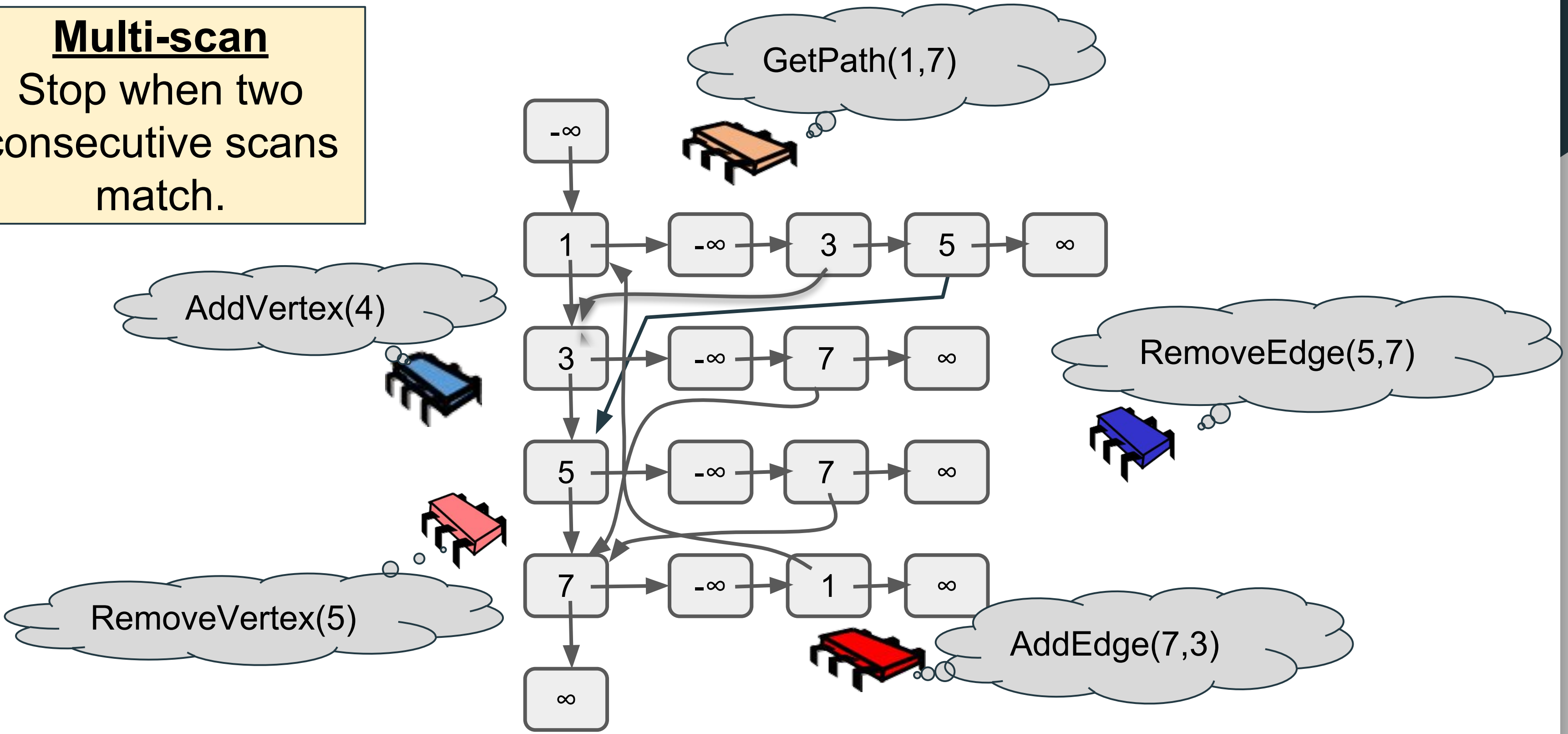
Reachability Query



Reachability Query

Multi-scan

Stop when two consecutive scans match.



Correctness

Theorem 1: *The ADT operations are* ***linearizable.***

Progress Guarantee

Theorem 2: The ADT operations are **non-blocking:**

1. *If the set of keys is finite, the operations ContainsVertex and ContainsEdge are wait-free.*

2. *The operation GetPath is obstruction-free.*

3. *The operations AddVertex, RemoveVertex, ContainsVertex, AddEdge, RemoveEdge, and ContainsEdge are lock-free.*

Experimental Setup

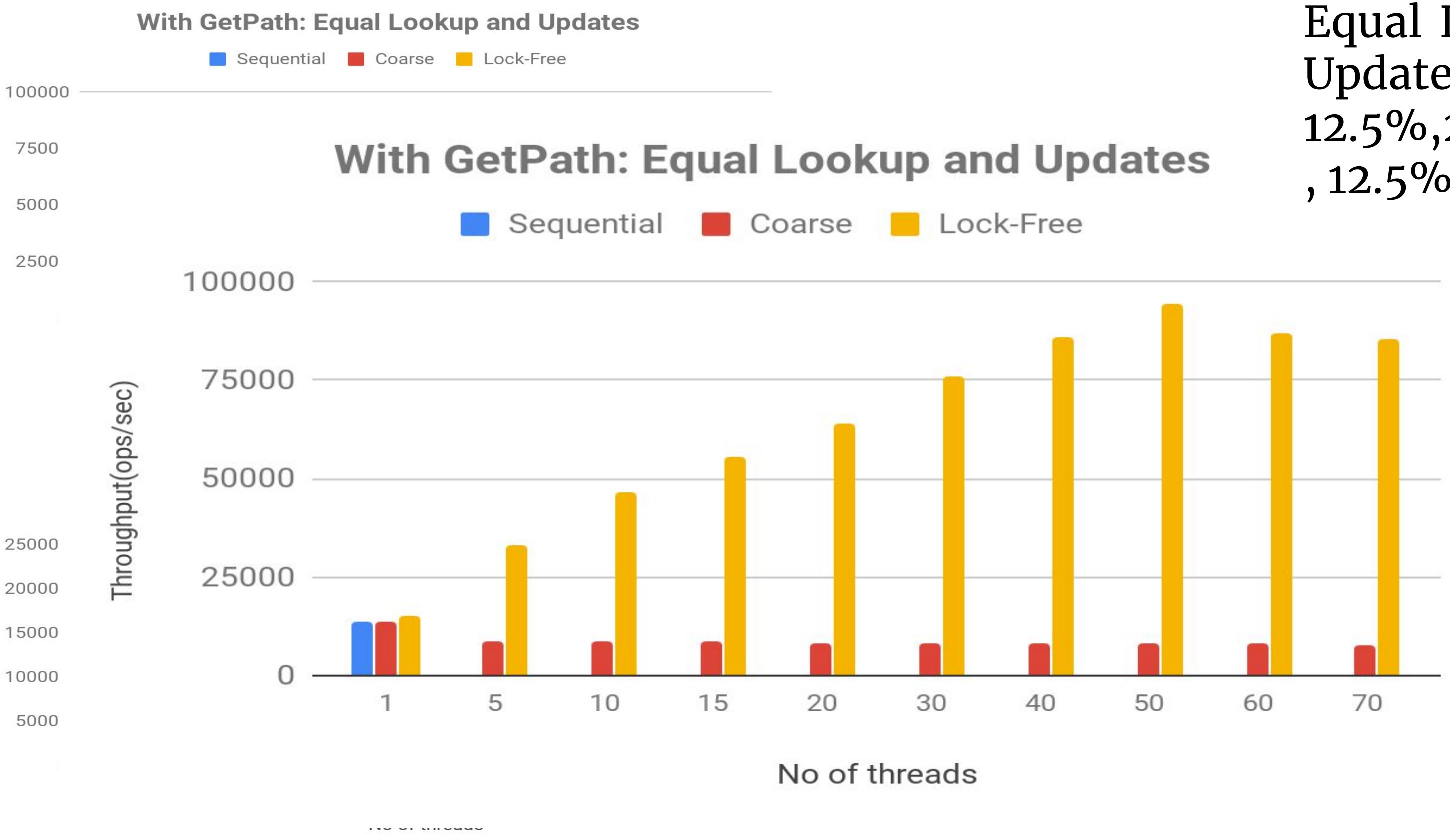
- ➔ Intel(R) Xeon(R) E5-2690 v4 CPU containing 56 cores running at 2.60GHz. and each core supports 2 logical threads.
- ➔ Implementation has been done in C++ (without any garbage collection) and multi-threaded implementation is based on Posix threads.
- ➔ Start experiments, with 1000 vertices and approximately 125000 edges added randomly.
- ➔ We measure throughput obtained on running the experiment for 20 seconds.
- ➔ Each data point is obtained by averaging over 5 iterations.
- ➔ We compare the non-blocking graph with its sequential and coarse-grained counterparts in two separate sets of experiments comprising:
    - (a) The ADT operations excluding GetPath.
    - (b) All the ADT operations.

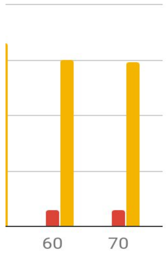
Workload Distributions With GetPath

With GetPath: {AddVertex, RemoveVertex, ContainsVertex, AddEdge, RemoveEdge, ContainsEdge, GetPath }

A. Equal Lookup and Updates: (12.5%, 12.5%, 24%,12.5%, 12.5%, 24%, 2%)
B. Lookup Intensive: (2%, 2%, 45%, 2%, 2%, 45%, 2%)
C. Update Intensive: (22.5%, 22.5%, 4%, 22.5%, 22.5%, 4%, 2%)

Equal Lookup and Updates:(12.5%, 12.5%,24%,12.5% , 12.5%, 24%, 2%)





Conclusion

- A novel concurrent directed Graph data structure represented by its adjacency list which can grow without bound and sink at the runtime.
- A simple and efficient non-blocking implementation of the ADT operations.

- The spotlight of our work is an obstruction-free reachability query.
- Provably all the methods are linearizable.

- We extensively evaluate a sample C/C++ implementation of the algorithm through a number of micro-benchmarks.

- Our experiments show that the proposed algorithm scales 5-7X with the number of threads in commonly available multi-core systems.

For More Information

The Technical Report is available at: URL: *https://arxiv.org/abs/1809.00896*
And the complete source code is available at: *https://github.com/PDCRL/ConcurrentGraphDS*

Thank You!

References

1. Bapi Chatterjee, Sathya Peri, Muktikanta Sa, and Nandini Singhal. *A Simple and Practical Concurrent Non-blocking Unbounded Graph with Linearizable Reachability Queries*. 20th ICDCN. 2019.
2. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press. 2009.
3. Timothy L. Harris. *A Pragmatic Implementation of Non-blocking Linked Lists*. 15th DISC. 2001.
4. Maurice Herlihy, Victor Luchangco, and Mark Moir. *Obstruction-Free Synchronization: Double-Ended Queues as an Example.* 23rd ICDCS. 2003.
5. Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.
6. Maurice P. Herlihy and Jeannette M. Wing. *Linearizability: a correctness condition for concurrent objects*. ACM Trans. Program. Lang. Syst. 1990.
7. Nikolaos D. Kallimanis and Eleni Kanellou. *Wait-Free Concurrent Graph Objects with Dynamic Traversals*. 19th OPODIS. 2015.

For More Information

The Technical Report is available at: URL: *https://arxiv.org/abs/1809.00896*
And the complete source code is available at: *https://github.com/PDCRL/ConcurrentGraphDS*

Node Structure

Edge Node

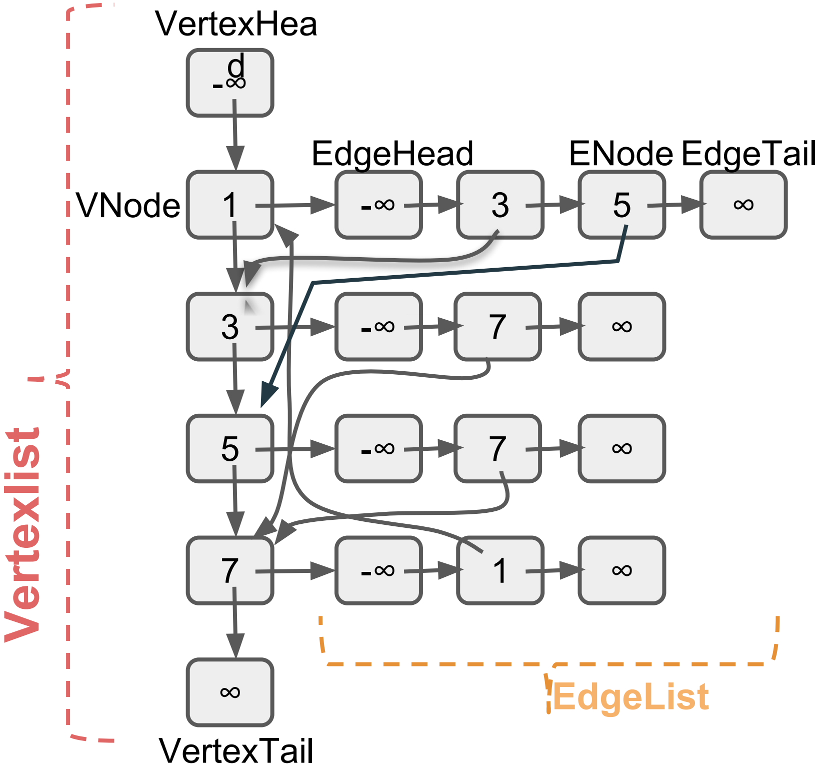
```
class ENode{
  int k;
  ENode enxt;
  VNode ptv;
};
```

Vertex Node

```
class VNode{
 int k;
 VNode vnxt;
 ENode enxt;
 int VisitedArray[];
 int ecnt;
};
```

BFS Node

```
class BFSNode{
 VNode n;
 BFSNode nxt;
 BFSNode p;
 Int locCnt;
};
```



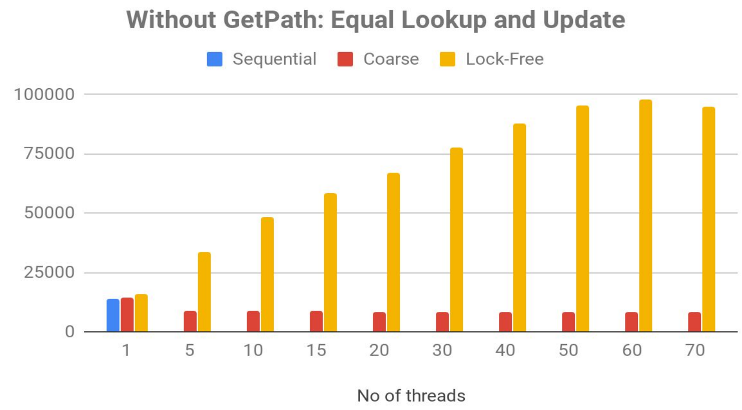
Observations

A. The non-blocking algorithm is scalable with the number of threads in the system.
B. The performance of lock-based algorithm degrades with the increasing number of threads.
C. 5x-7x increase in the throughput in comparison to the sequential and lock-based counterparts.

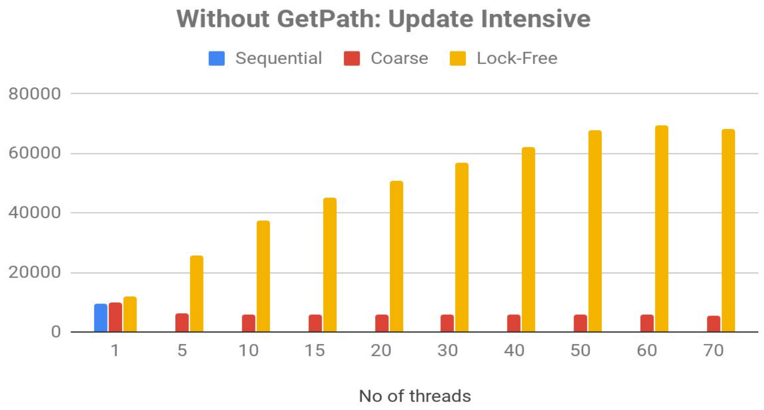
Workload Distributions Without GetPath

Without GetPath: {AddVertex, RemoveVertex, ContainsVertex, AddEdge, RemoveEdge, ContainsEdge }

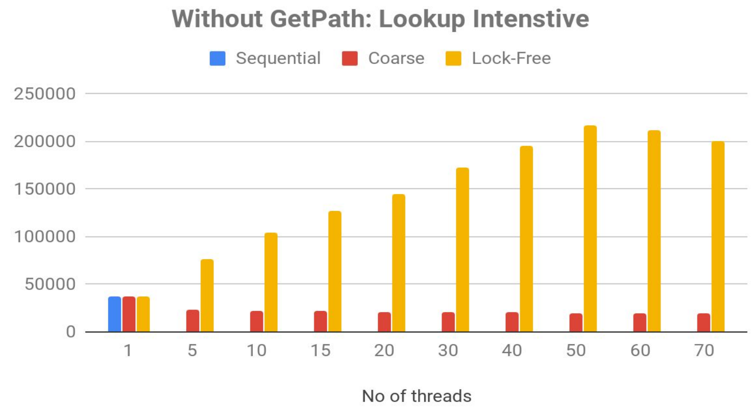
A. Equal Lookup and Updates: (12.5%, 12.5%, 25%, 12.5%, 12.5%, 25%)
B. Lookup Intensive: (2.5%, 2.5%, 45%, 2.5%, 2.5%, 45%)
C. Update Intensive: (22.5%, 22.5%, 5%, 22.5%, 22.5%, 5%)



Without GetPath: Equal Lookup and Update

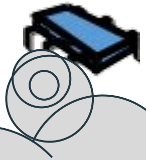


Without GetPath: Update Intensive



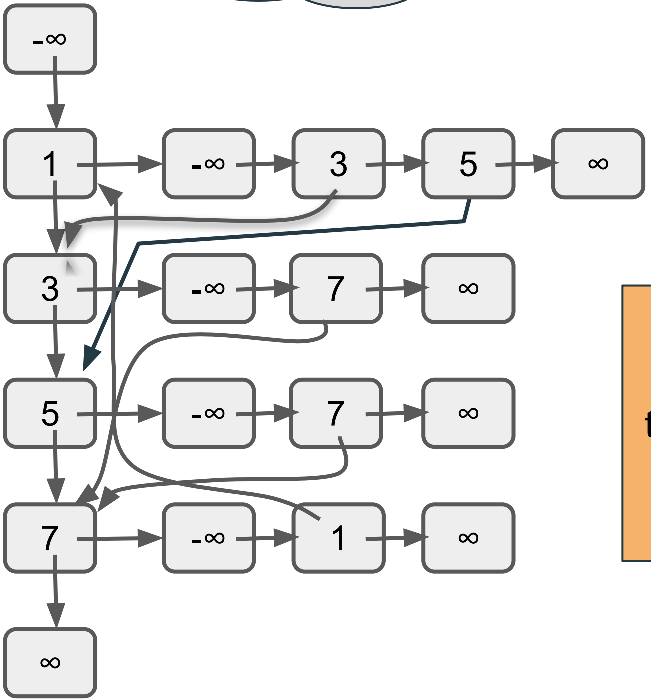
Without GetPath: Lookup Intenstive

Reachability Query



After successfully checking of v(1) and v(7), it performs repeated BFS traversals by invoking the procedure **Scan**

GetPath(1,7)



During any edge modification operations the atomic counter ***ecnt*** of the source vertex is necessarily incremented