# Building Efficient Concurrent Graph Object through Composition of List-based Set

Sathya Peri  Muktikanta Sa  **Nandini Singhal**

Department of Computer Science & Engineering
Indian Institute of Technology Hyderabad

AADDA Workshop in Conjunction with ICDCN 2018

January 4, 2018

# Outline of the Presentation

1. Motivation
2. Problem Definition
3. Our Methodology
4. Working of the methods
5. Correctness
6. Empirical Results
7. Conclusion & Future Work

# Outline of the Presentation

# Graphs are Everywhere...

- Common real world objects can be modeled as graphs, which build the pairwise relations between objects.
- Graphs are used in the fields: genomics, networks, coding theory, scheduling, computational devices, networks, organization of similar and dissimilar objects, etc.
- Day by day the size of the above graphs are increasing exponentially.
- Generally, these graphs are very *large* and *dynamic* in nature.

# Graphs are Everywhere...

- Common real world objects can be modeled as graphs, which build the pairwise relations between objects.
- Graphs are used in the fields: genomics, networks, coding theory, scheduling, computational devices, networks, organization of similar and dissimilar objects, etc.
- Day by day the size of the above graphs are increasing exponentially.
- Generally, these graphs are very *large* and *dynamic* in nature.
- **Fully Dynamic Graphs** allow both insertions and deletions.

# Need for a Concurrent Graph Data Structure

Processes acting on a dynamic graph needs to update it frequently and run computations.

# Need for a Concurrent Graph Data Structure

Processes acting on a dynamic graph needs to update it frequently and run computations.

1. If global lock is used, then global bottleneck
2. Partition the graph into disjoint sets. Any update to the graph leads to re-partitioning $\rightarrow$ expensive!

# Need for a Concurrent Graph Data Structure

Processes acting on a dynamic graph needs to update it frequently and run computations.

1. If global lock is used, then global bottleneck
2. Partition the graph into disjoint sets. Any update to the graph leads to re-partitioning → expensive!

Need for Independent access to disjoint parts of graph.

# Outline of the Presentation

# Problem Statement

Given a initial graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of edges. Threads can perform *six* basic operations:

# Problem Statement

Given a initial graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of edges. Threads can perform *six* basic operations:

1. add_Vertex(v)
2. add_Edge(u, v)

# Problem Statement

Given a initial graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of edges. Threads can perform *six* basic operations:

1. add_Vertex(v)
2. add_Edge(u, v)
3. delete_Vertex(v)
4. delete_Edge(u, v)

# Problem Statement

Given a initial graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of edges. Threads can perform *six* basic operations:

1. add_Vertex(v)
2. add_Edge(u, v)
3. delete_Vertex(v)
4. delete_Edge(u, v)
5. ContainsEdge(u, v)
6. ContainsVertex(u)

# Problem Statement

Given a initial graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of edges. Threads can perform *six* basic operations:

1. add_Vertex(v)
2. add_Edge(u, v)
3. delete_Vertex(v)
4. delete_Edge(u, v)
5. ContainsEdge(u, v)
6. ContainsVertex(u)

Note: This is a *directed* unweighted simple graph.

# Difficulties with Fully Dynamic Graphs

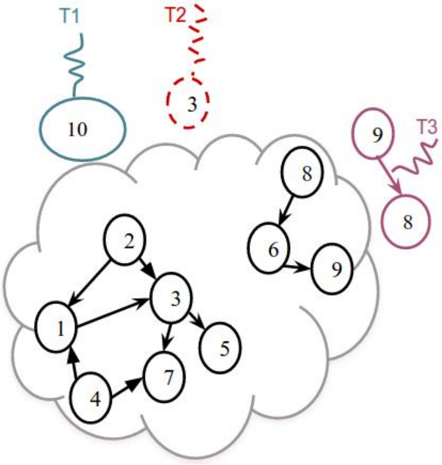



Figure : Thread $T_1$ & $T_3$ adding the vertex 10 and the edge$(9, 8)$ respectively, on the other hand the thread $T_2$ wants to delete the vertex 3.

# Dynamic Graphs

- Dynamic graph algorithms perform better than their static counterparts because of increased data parallelism.

# Dynamic Graphs

- Dynamic graph algorithms perform better than their static counterparts because of increased data parallelism.
- However proving the correctness is more challenging as they allow concurrent access at a finer granularity and access common data items.

# Outline of the Presentation

1. Motivation
2. Problem Definition
3. **Our Methodology**
4. Working of the methods
5. Correctness
6. Empirical Results
7. Conclusion & Future Work

# Contribution

Representation of concurrent directed graph data structure as an adjacency list which has been implemented as a concurrent set based on linked list. *[Steve Heller, et al.]*

# Methods Exported

1. *AddVertex(u)* adds a vertex $u$ to $G$, returning *true* iff vertex $u$ was not already in $G$ else *false*.

# Methods Exported

1. *AddVertex(u)* adds a vertex $u$ to $G$, returning *true* iff vertex $u$ was not already in $G$ else *false*.
2. *RemoveVertex(u)* deletes vertex $u$ from $G$, returning *true* iff $u$ was present else returns *false*.

# Methods Exported

1. *AddVertex(u)* adds a vertex $u$ to $G$, returning *true* iff vertex $u$ was not already in $G$ else *false*.
2. *RemoveVertex(u)* deletes vertex $u$ from $G$, returning *true* iff $u$ was present else returns *false*.
3. *AddEdge(u, v)* adds a directed edge $(u, v)$ to the concurrent $G$, returning *true* iff $(u, v)$ was not already present in $G$ else returns *false*.

# Methods Exported

1. *AddVertex(u)* adds a vertex $u$ to $G$, returning *true* iff vertex $u$ was not already in $G$ else *false*.
2. *RemoveVertex(u)* deletes vertex $u$ from $G$, returning *true* iff $u$ was present else returns *false*.
3. *AddEdge(u, v)* adds a directed edge $(u, v)$ to the concurrent $G$, returning *true* iff $(u, v)$ was not already present in $G$ else returns *false*.
4. *RemoveEdge(u,v)* deletes the directed edge $(u, v)$ from $G$, returning *true* iff $(u, v)$ was already there else returns *false*.

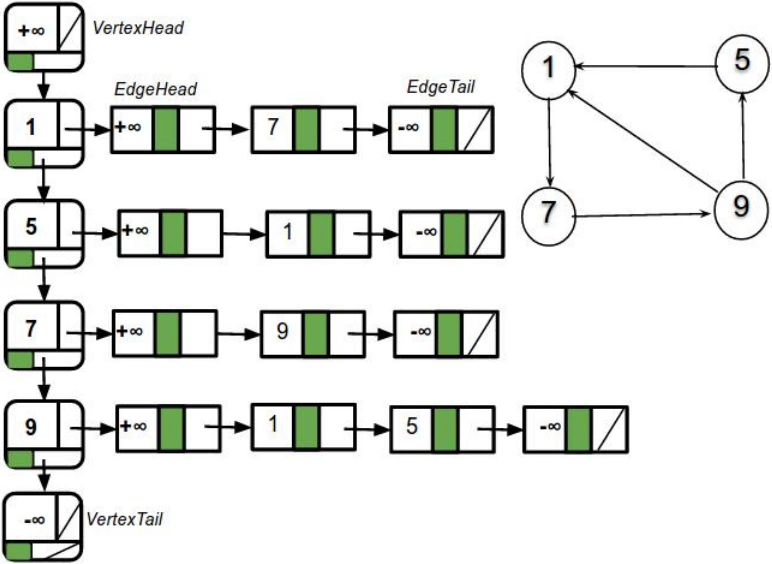
# Methods Exported

1. *AddVertex(u)* adds a vertex $u$ to $G$, returning *true* iff vertex $u$ was not already in $G$ else *false*.
2. *RemoveVertex(u)* deletes vertex $u$ from $G$, returning *true* iff $u$ was present else returns *false*.
3. *AddEdge(u, v)* adds a directed edge $(u, v)$ to the concurrent $G$, returning *true* iff $(u, v)$ was not already present in $G$ else returns *false*.
4. *RemoveEdge(u,v)* deletes the directed edge $(u, v)$ from $G$, returning *true* iff $(u, v)$ was already there else returns *false*.
5. *ContainsEdge(u,v)* returns *true* iff $G$ contains the edge $(u, v)$ else returns *false*.

# Methods Exported

1. *AddVertex(u)* adds a vertex $u$ to $G$, returning *true* iff vertex $u$ was not already in $G$ else *false*.
2. *RemoveVertex(u)* deletes vertex $u$ from $G$, returning *true* iff $u$ was present else returns *false*.
3. *AddEdge(u, v)* adds a directed edge $(u, v)$ to the concurrent $G$, returning *true* iff $(u, v)$ was not already present in $G$ else returns *false*.
4. *RemoveEdge(u,v)* deletes the directed edge $(u, v)$ from $G$, returning *true* iff $(u, v)$ was already there else returns *false*.
5. *ContainsEdge(u,v)* returns *true* iff $G$ contains the edge $(u, v)$ else returns *false*.
6. *ContainsVertex(u)* returns *true* iff $G$ contains the vertex $u$ else returns *false*.

# Construction of Concurrent List based Directed Graph



# Concurrent List-based Set

Set implemented using linked-list, a collection of items that contains no duplicate elements and exported methods are:

1. **add(x)**: adds x to the set, returning true if, and only if x was not already present earlier.
2. **remove(x)**: removes x from the set, returning true if, and only if x was there.
3. **contains(x)**: returns true if, and only if the set contains x.

# Variants

1. **Sequential**: Only one thread and No Lock.
2. **Coarse-grained synchronization**: Uses Single Spin Lock.
3. **Fine-grained synchronization**: Split the object into independently synchronized components.
4. **Optimistic synchronization**: Search without acquiring any locks.
5. **Lazy synchronization**: Postpone the hard work, a node has a bool marked field: logically removal (setting a marked bit) and physical removal (unlinking).
6. **Non-blocking synchronization**: No locks and use the built-in atomic operations compareAndSet() for synchronization.

# Correctness and Progress Conditions

Designing of any method or data-structure in the concurrent world, needs to satisfy these two properties: *[Maurice Herlihy, et al]*

# Correctness and Progress Conditions

Designing of any method or data-structure in the concurrent world, needs to satisfy these two properties: *[Maurice Herlihy, et al]*

1. Correctness and Safety: **Linearizability**

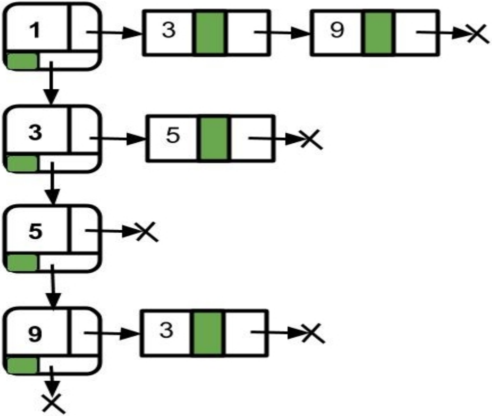
# Correctness and Progress Conditions

Designing of any method or data-structure in the concurrent world, needs to satisfy these two properties: *[Maurice Herlihy, et al]*

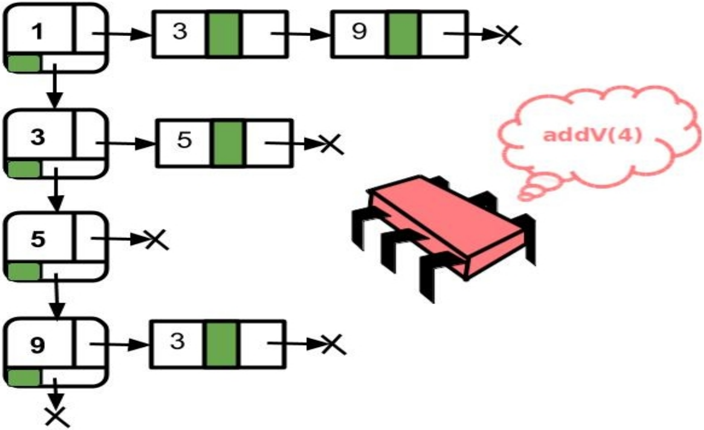
1. Correctness and Safety: **Linearizability**
2. Liveness: **Progress Conditions**

# Outline of the Presentation

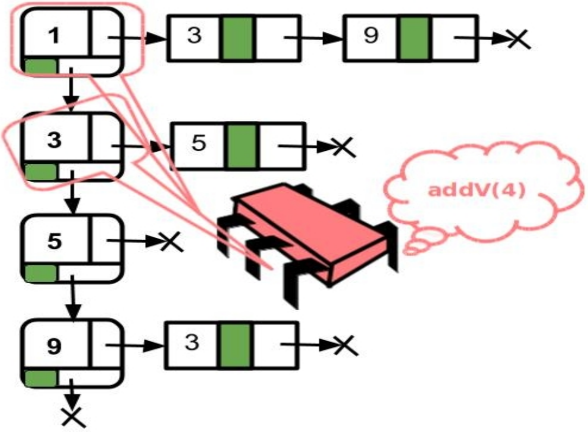
# Working of AddVertex(u) method



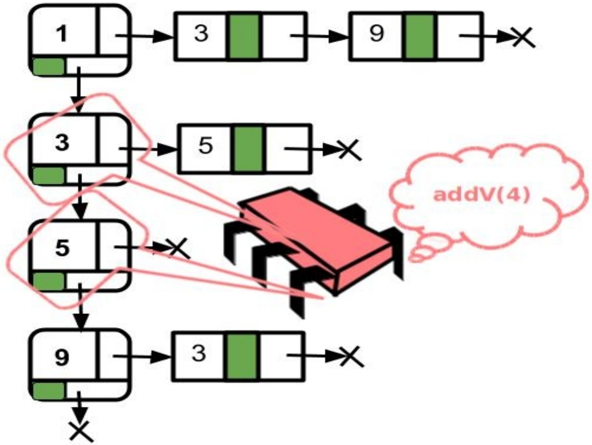
# Working of AddVertex(u) method



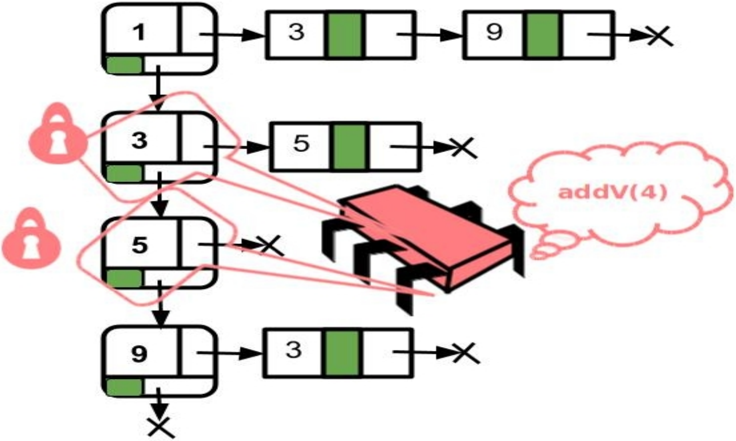
# Working of AddVertex(u) method



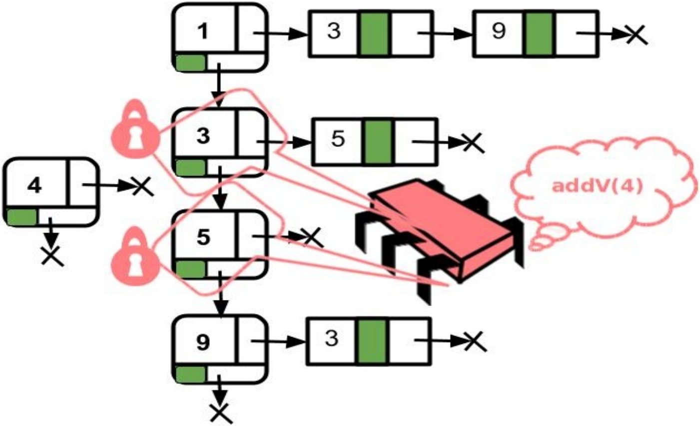
# Working of AddVertex(u) method



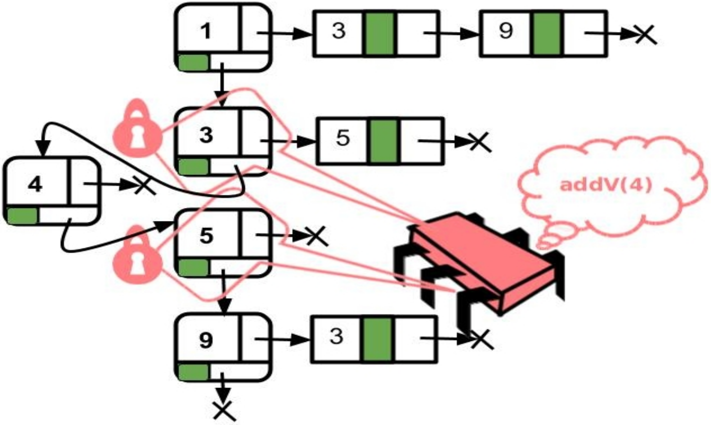
# Working of AddVertex(u) method



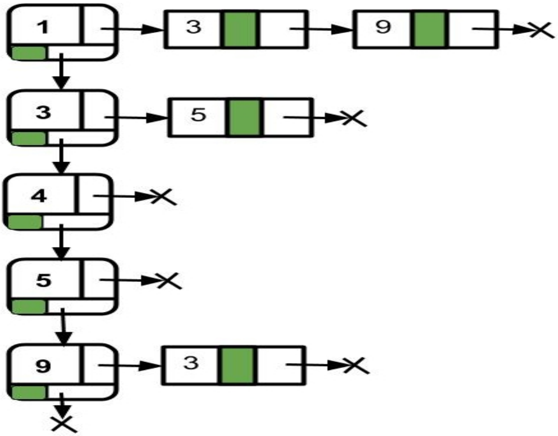
# Working of AddVertex(u) method



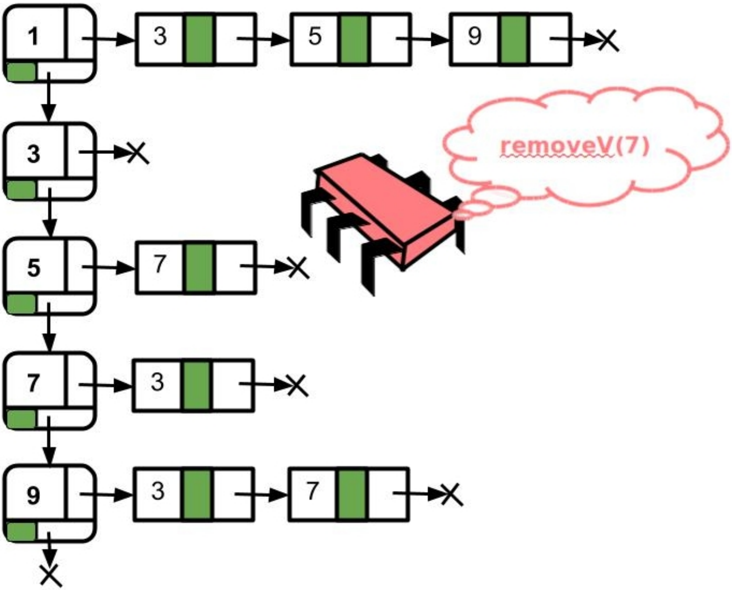
# Working of AddVertex(u) method



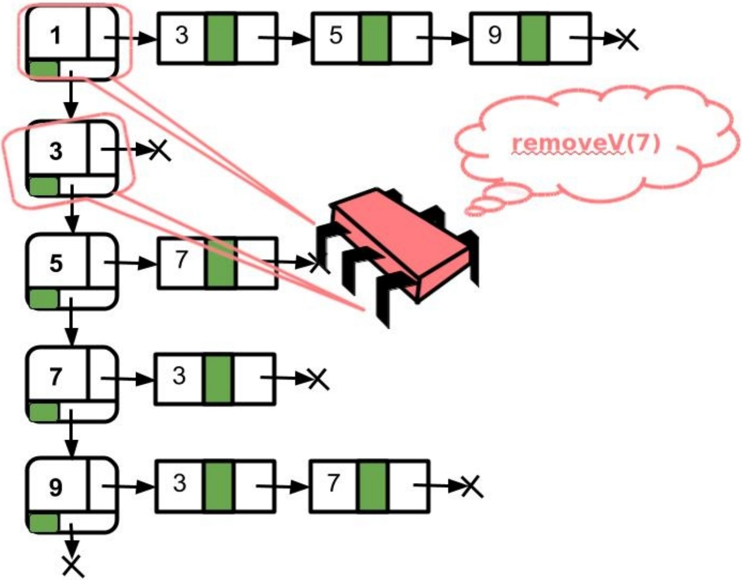
# Working of AddVertex(u) method



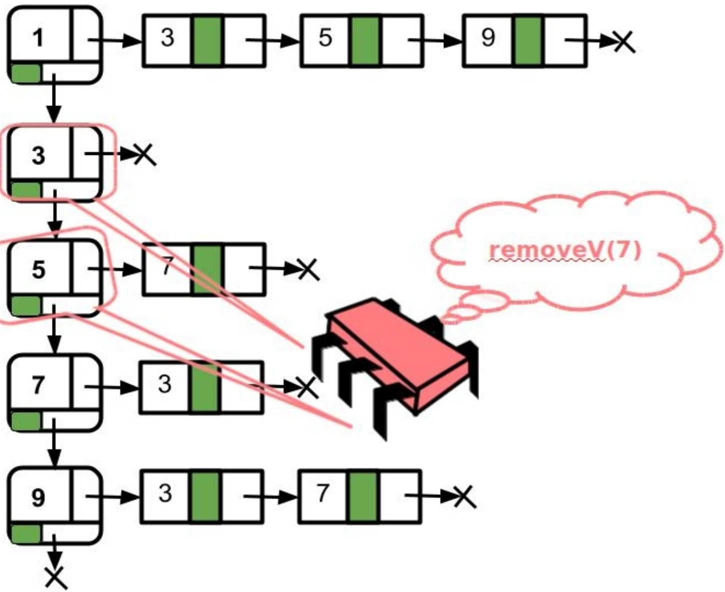
# Working of RemoveVertex(u) method



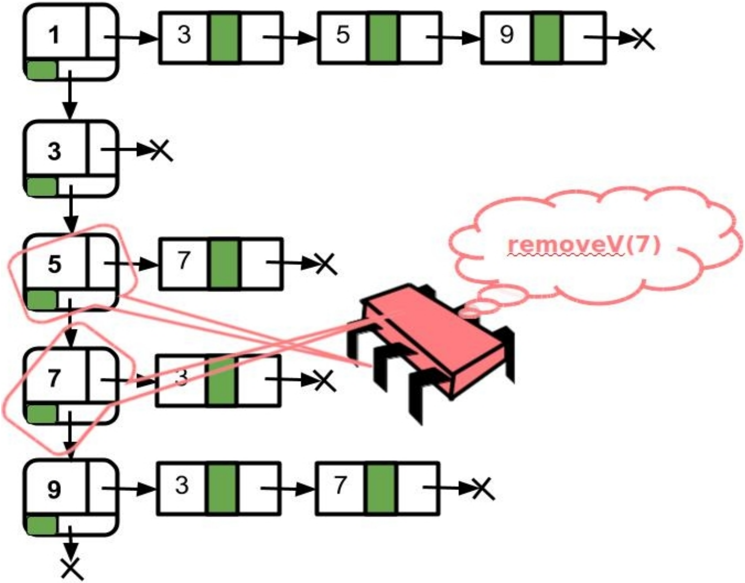
# Working of RemoveVertex(u) method



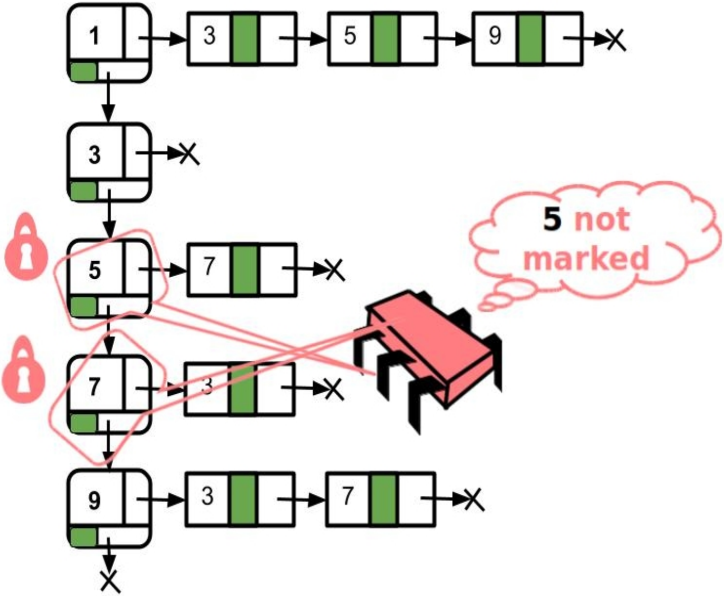
# Working of RemoveVertex(u) method



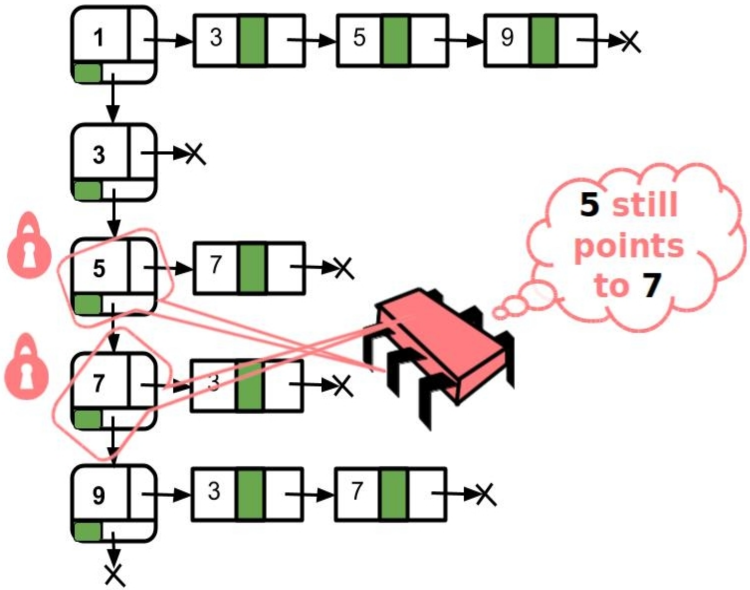
# Working of RemoveVertex(u) method



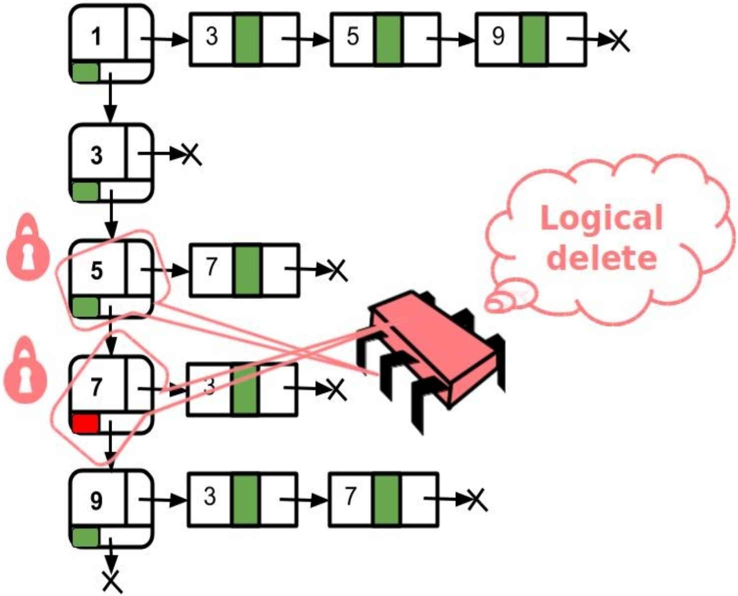
# Working of RemoveVertex(u) method



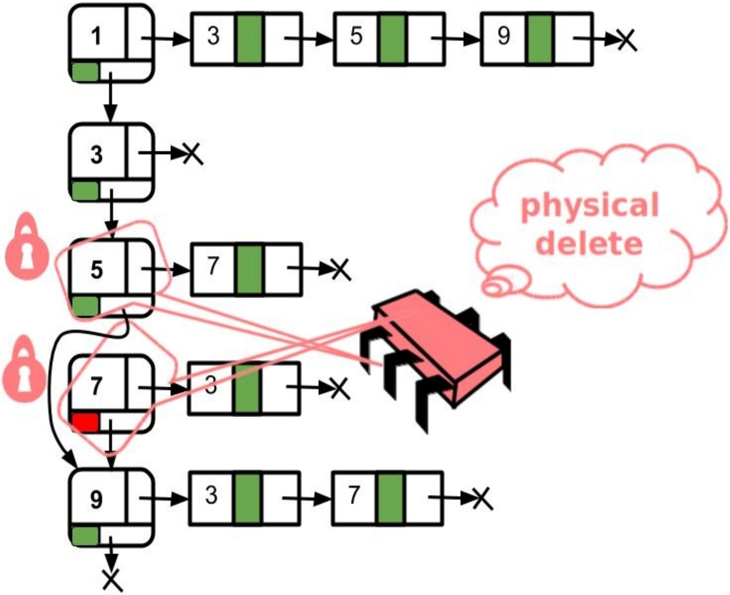
# Working of RemoveVertex(u) method



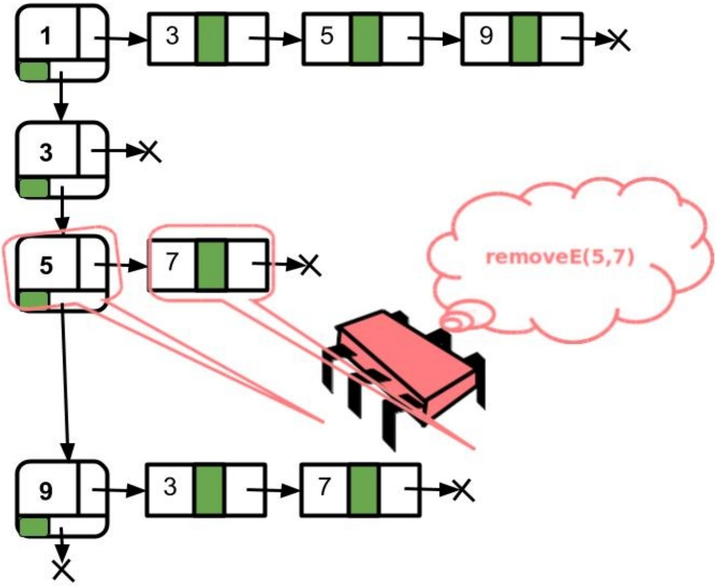
# Working of RemoveVertex(u) method



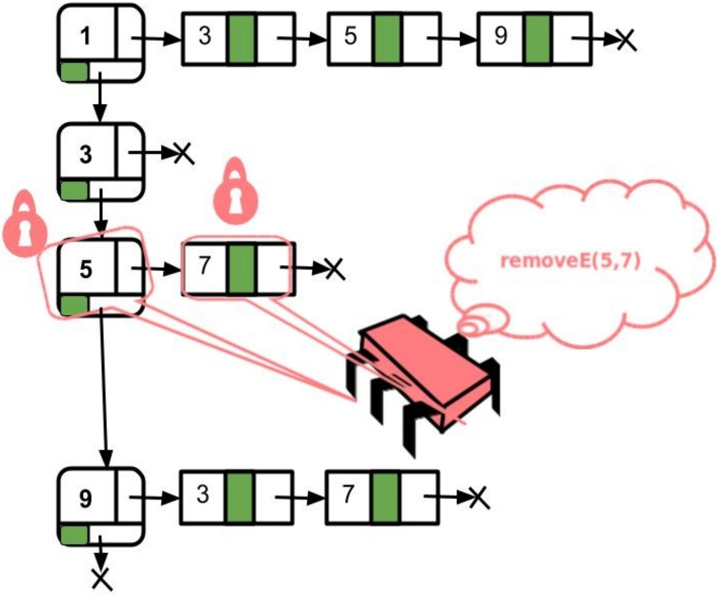
# Working of RemoveVertex(u) method



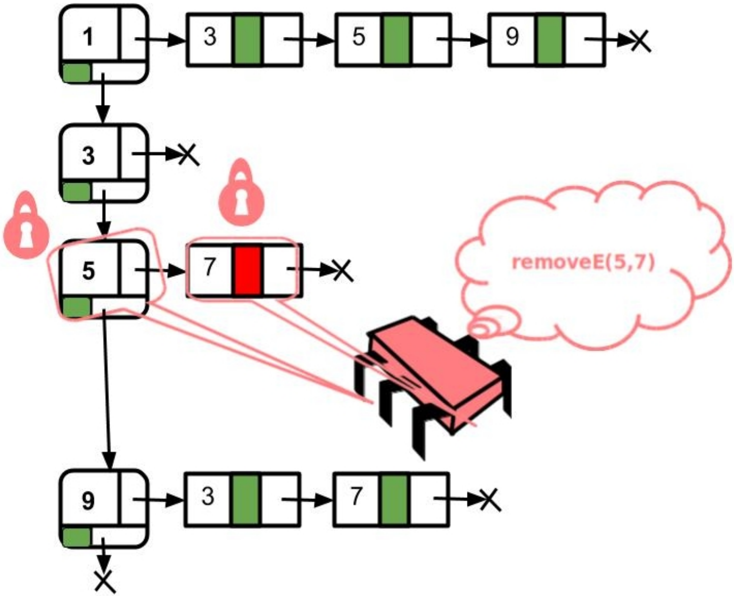
# Working of RemoveVertex(u) method



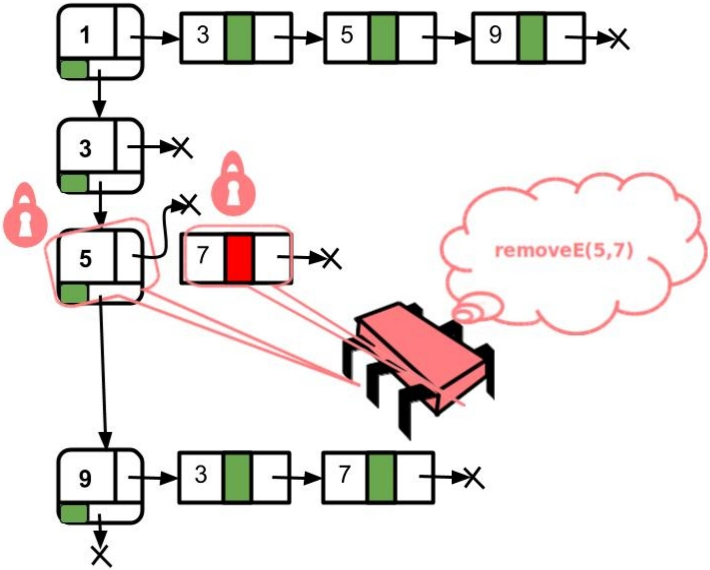
# Working of RemoveVertex(u) method



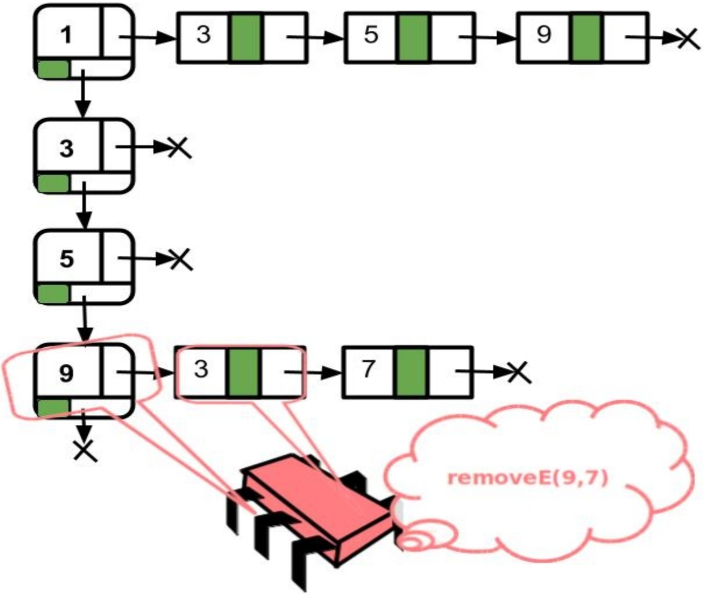
# Working of RemoveVertex(u) method



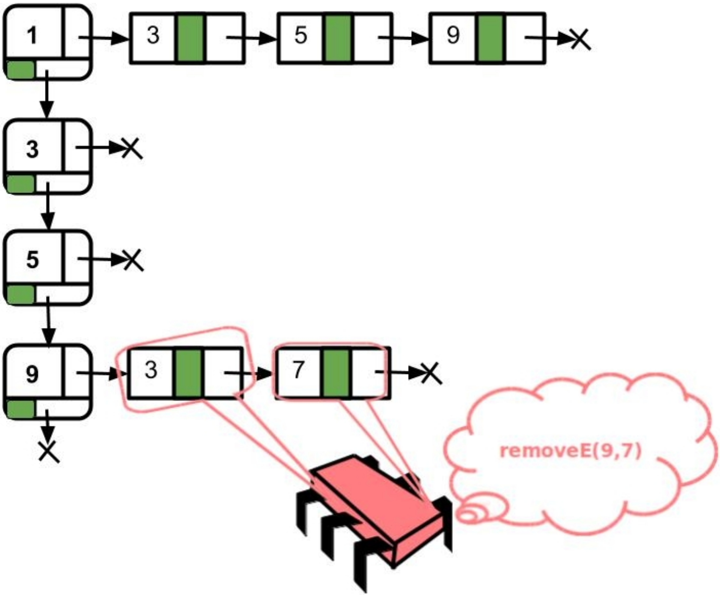
# Working of RemoveVertex(u) method



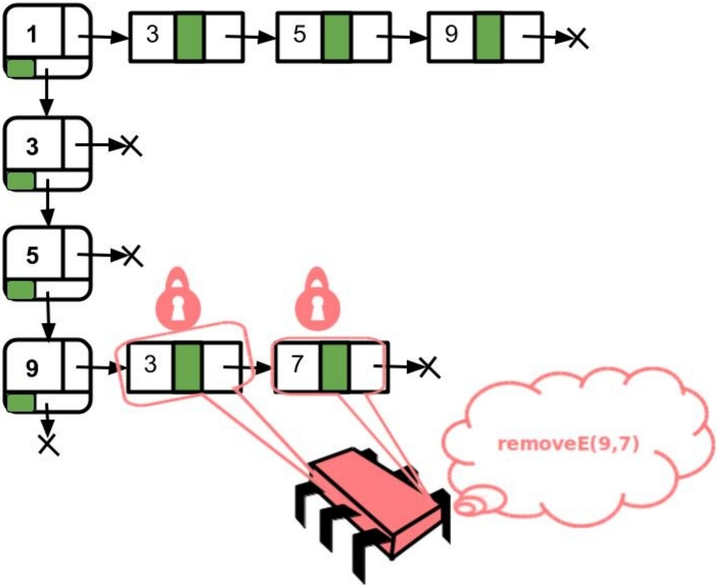
# Working of RemoveVertex(u) method



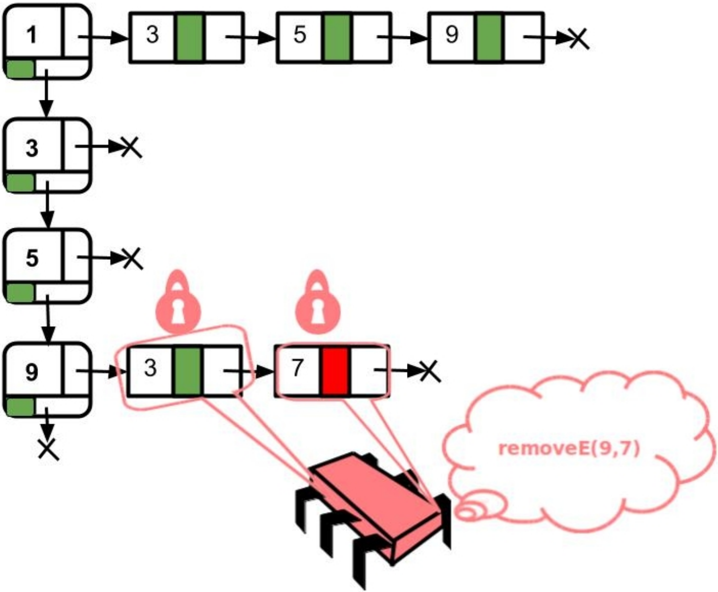
# Working of RemoveVertex(u) method



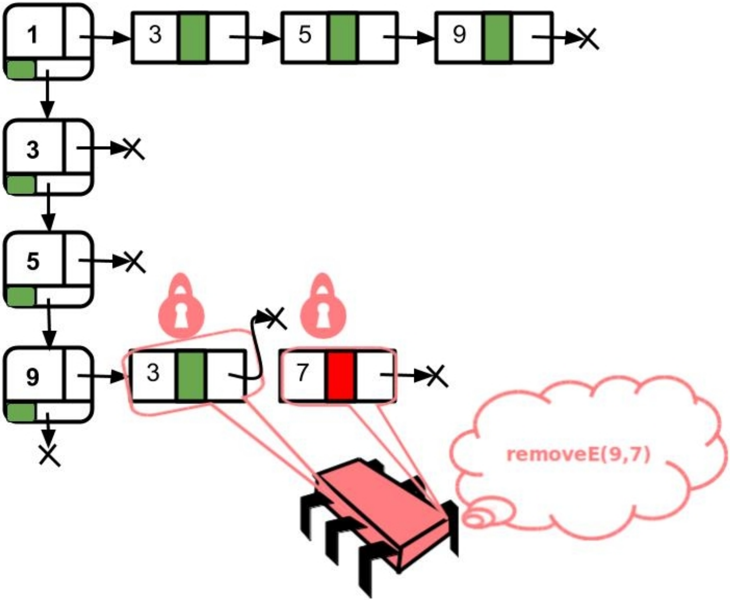
# Working of RemoveVertex(u) method



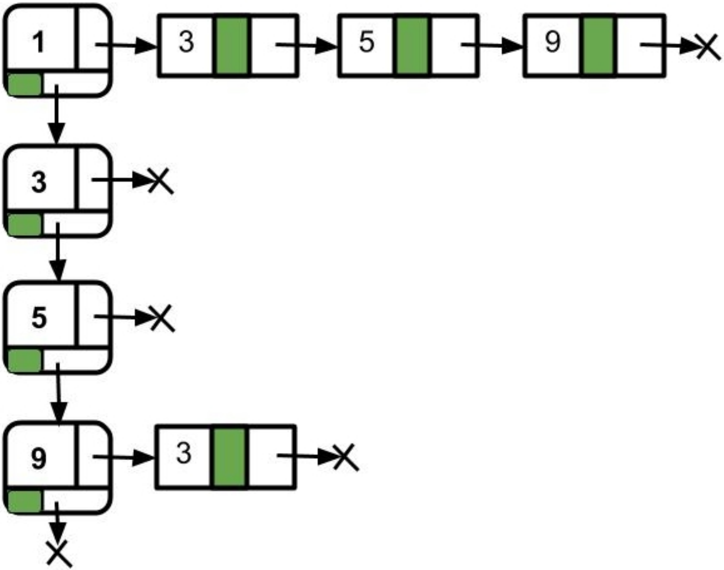
# Working of RemoveVertex(u) method



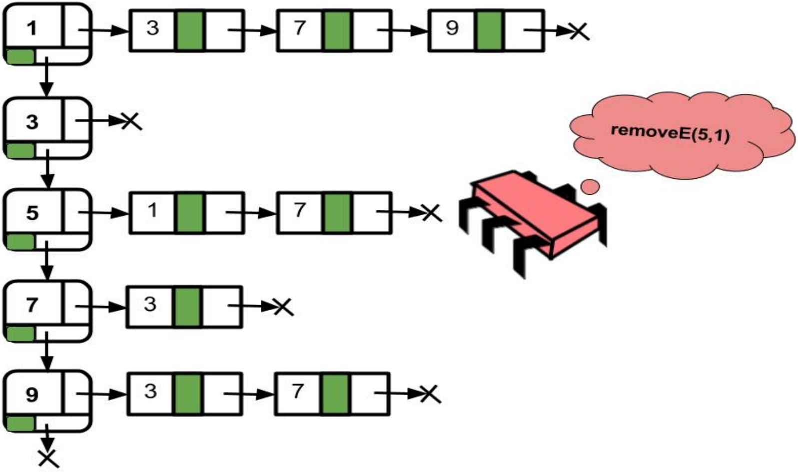
# Working of RemoveVertex(u) method



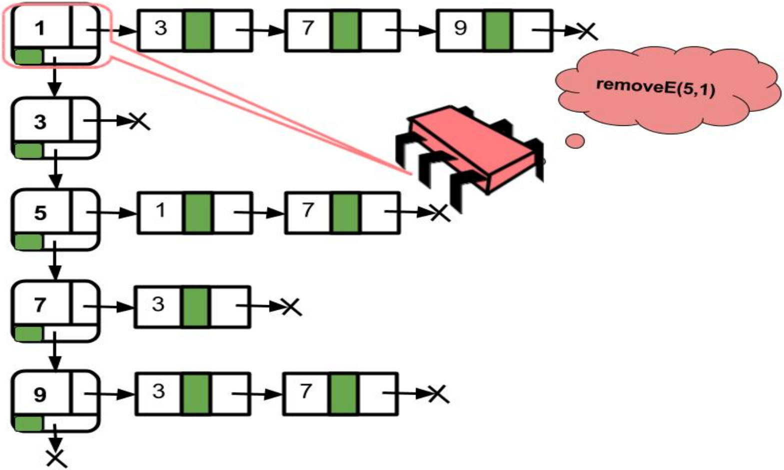
# Working of RemoveVertex(u) method



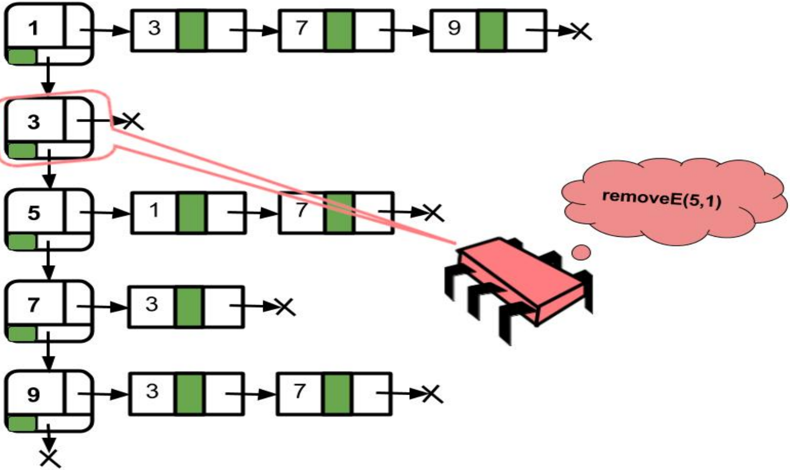
# Working of RemoveEdge(u, v) method



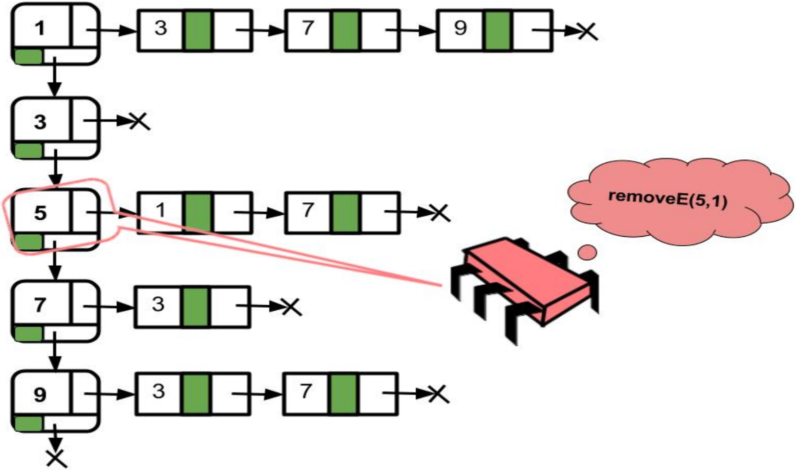
# Working of RemoveEdge(u, v) method



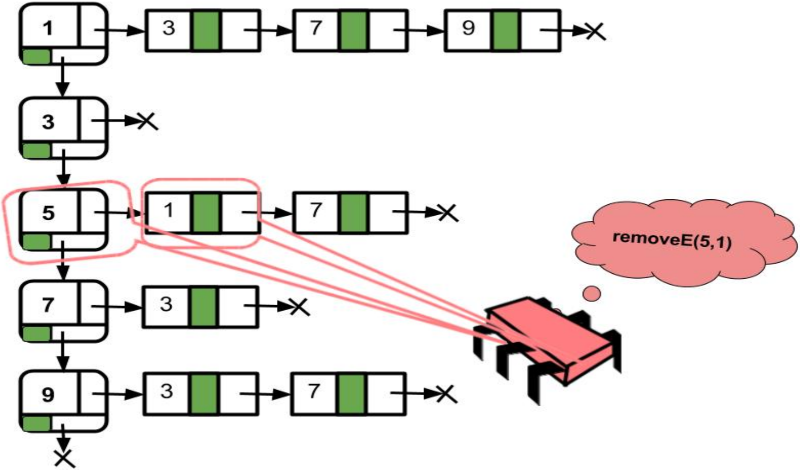
# Working of RemoveEdge(u, v) method



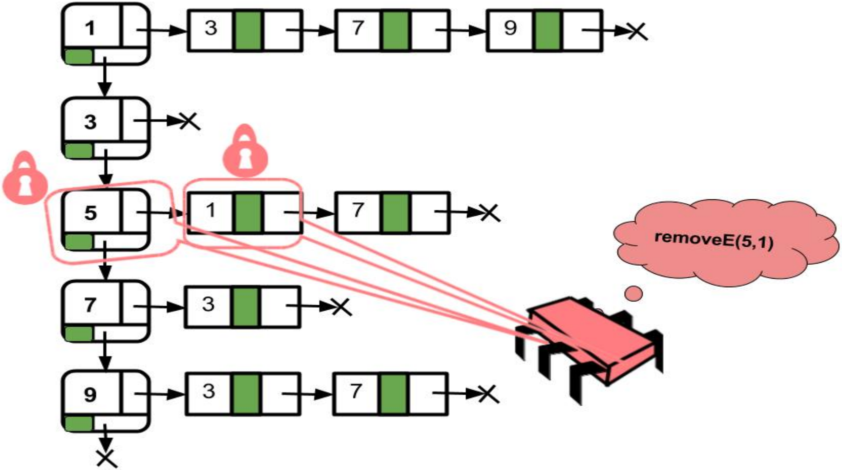
# Working of RemoveEdge(u, v) method



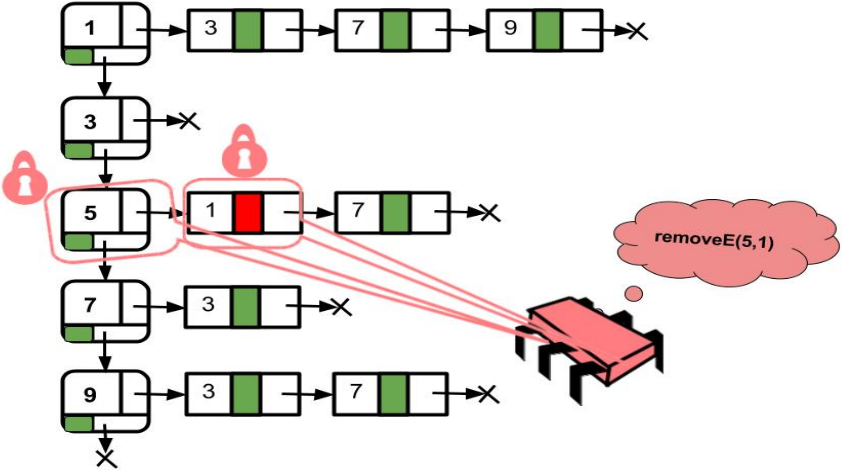
# Working of RemoveEdge(u, v) method



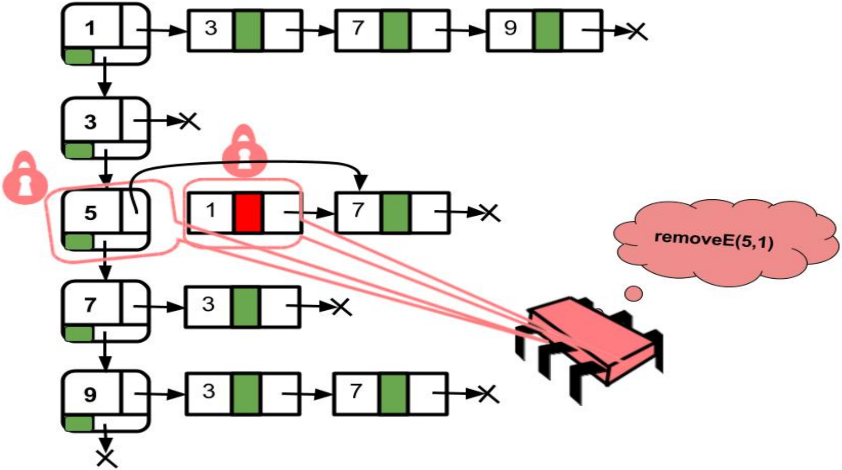
# Working of RemoveEdge(u, v) method



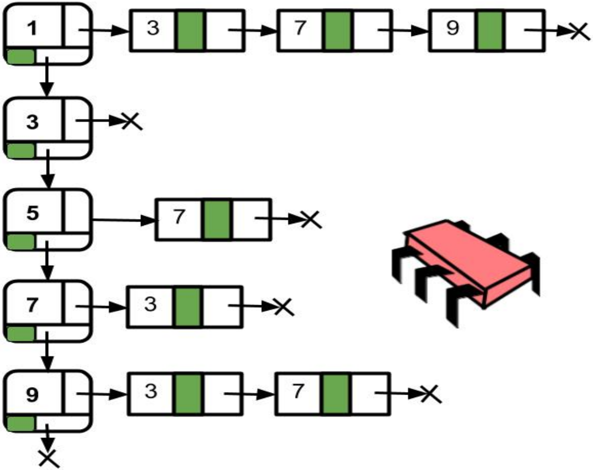
# Working of RemoveEdge(u, v) method



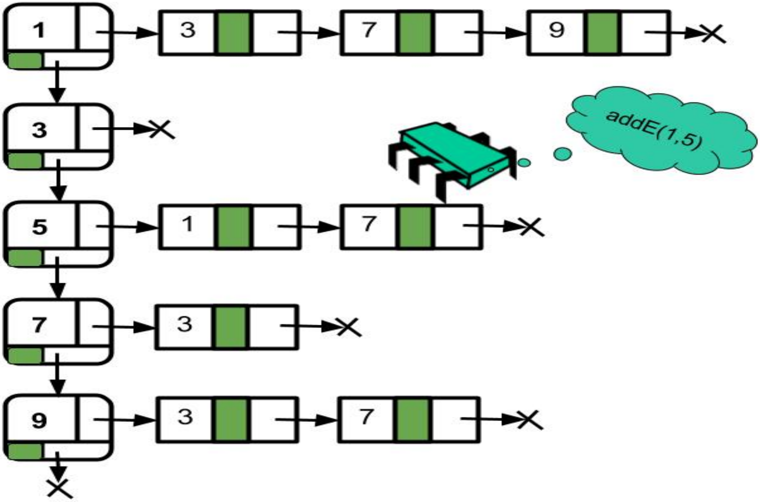
# Working of RemoveEdge(u, v) method



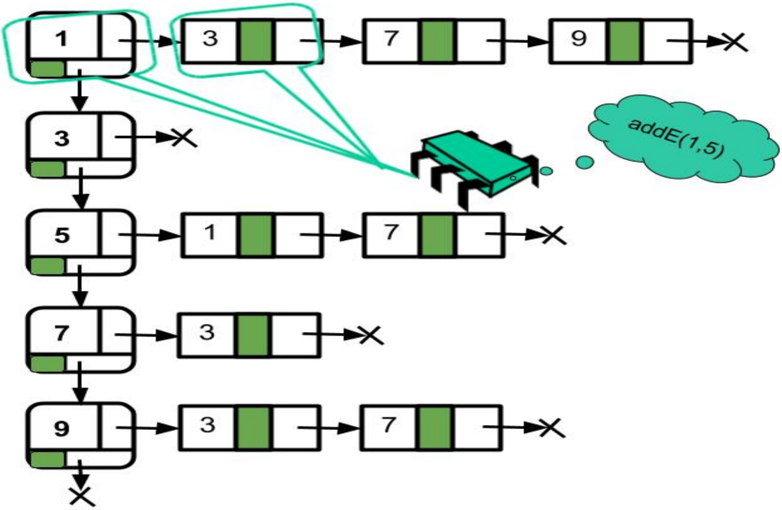
# Working of RemoveEdge(u, v) method



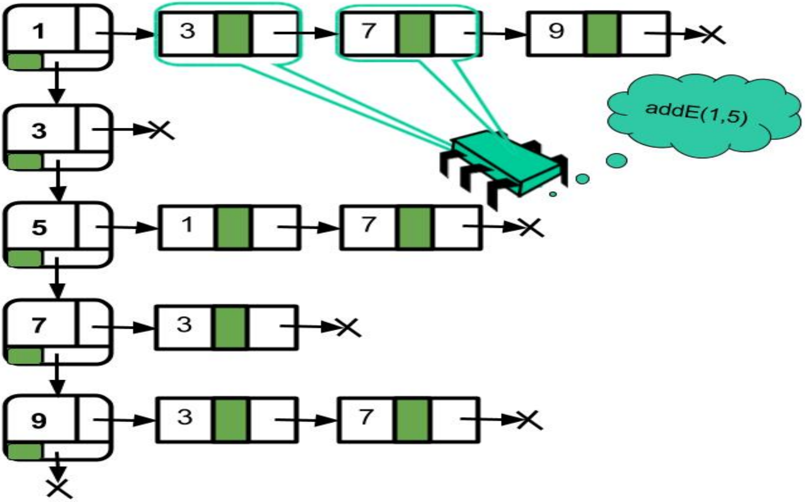
# Working of AddEdge(u, v) method



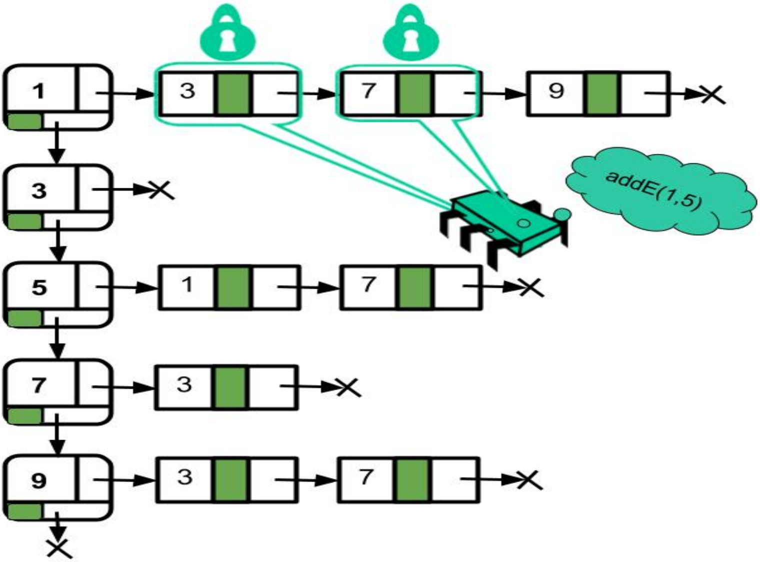
# Working of AddEdge(u, v) method



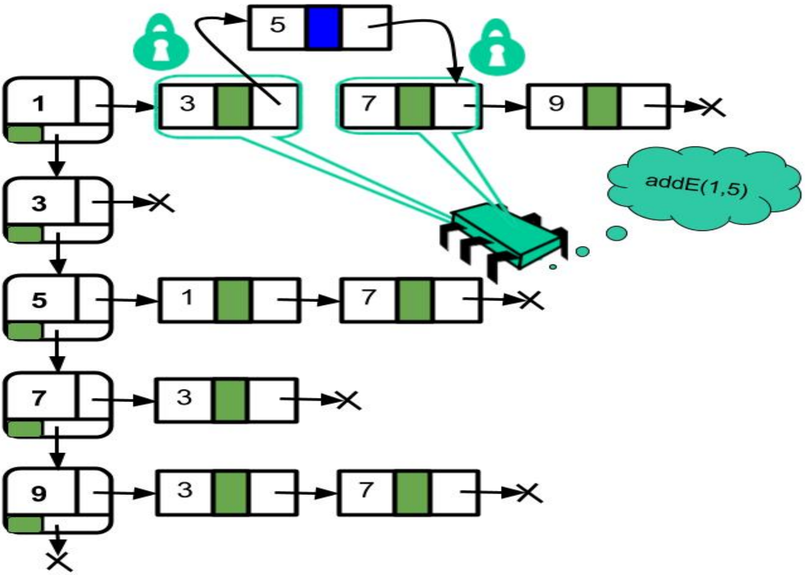
# Working of AddEdge(u, v) method



# Working of AddEdge(u, v) method



# Working of AddEdge(u, v) method



# Outline of the Presentation

# What is Linearizability?

# What is Linearizability?

- A history is a sequence of invocations and responses made of an object by a set of threads.
- Each invocation of a function will have a subsequent response.

A correctness condition for concurrent objects, by *[Maurice Herlihy, et al.]*

**Definition**

Each method call should appear to take effect instantaneously at some moment between its invocation and response.

# Linearizability Contd...

A history is linearizable if:

- its invocations and responses can be reordered to yield a sequential history;

# Linearizability Contd...

A history is linearizable if:

- its invocations and responses can be reordered to yield a sequential history;
- that sequential history is correct according to the sequential definition of the object;

# Linearizability Contd...

A history is linearizable if:

- its invocations and responses can be reordered to yield a sequential history;
- that sequential history is correct according to the sequential definition of the object;
- if a response preceded an invocation in the original history, it must still precede it in the sequential reordering.

# Example of Linearizability

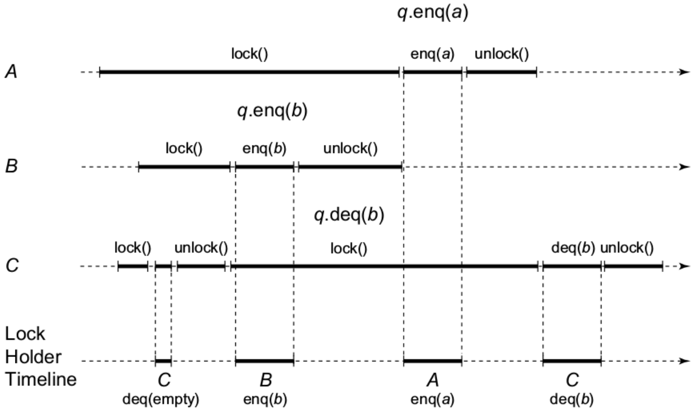



Figure : An execution of Concurrent Blocking queue with its linearization points

# Progress Guarantees

**Blocking**: In this, an arbitrary and unexpected delay by any thread (say, one holding a lock) can prevent other threads from making progress.

# Progress Guarantees

**Blocking**: In this, an arbitrary and unexpected delay by any thread (say, one holding a lock) can prevent other threads from making progress.

**Non-Blocking**: This condition ensures that threads competing for a shared resource do not have their execution indefinitely postponed by mutual exclusion.

# Blocking Progress Guarantees

**Deadlock-free:**

- A method is said to be deadlock-free, meaning that **some** thread trying to acquire the lock eventually succeeds.
- The system as a whole makes progress, but does not guarantee progress to individual threads.
- Weakest progress condition.

# Blocking Progress Guarantees

## Deadlock-free:

- A method is said to be deadlock-free, meaning that **some** thread trying to acquire the lock eventually succeeds.
- The system as a whole makes progress, but does not guarantee progress to individual threads.
- Weakest progress condition.

## Starvation-free:

- A method is starvation-free if **every** thread that attempts to acquire the lock eventually succeeds.

# Non-blocking Progress

An algorithm is **Non-blocking**: If failure or suspension of any thread cannot cause failure or suspension of another thread, for some operations.

A non-blocking algorithm can be

- *Lock-free*
- *Wait-free*
- *Obstruction-free*

# Non-Blocking Progress Guarantees Contd..

## Lock-freedom

- A method is lock-free if **some** thread that calls a method eventually returns.
- A lock-free data structure doesn't use any mutex locks.

# Non-Blocking Progress Guarantees Contd..

## Lock-freedom

- A method is lock-free if **some** thread that calls a method eventually returns.
- A lock-free data structure doesn't use any mutex locks.

## Wait-freedom

- A method is wait-free if **every** thread that calls that method eventually returns in a finite number of its steps.

# Non-Blocking Progress Guarantees Contd..

## Lock-freedom

- A method is lock-free if **some** thread that calls a method eventually returns.
- A lock-free data structure doesn't use any mutex locks.

## Wait-freedom

- A method is wait-free if **every** thread that calls that method eventually returns in a finite number of its steps.

## Obstruction-freedom

- A method is obstruction-free if every thread that calls that method returns if that thread executes in **isolation** for long enough.

# The Relationship among All

| | Non-Blocking | Blocking |
|---|---|---|
| Everyone makes progress | Wait-free | Starvation-free |
| Someone makes progress | Lock-free | Deadlock-free |

Figure : The Periodic Table of Progress Conditions

# Linearization Point of AddVertex(u)

If the method returns successfully (true),

1. Point where new vertex node is reachable from the head

# Linearization Point of AddVertex(u)

If the method returns successfully (true),

1. Point where new vertex node is reachable from the head

If the method returns unsuccessfully,

1. Point where a vertex node with same key is found in the vertex list

# Linearization Point of RemoveVertex(u)

If the method returns successfully (true),

1. Point where vertex node is logically marked as deleted

# Linearization Point of RemoveVertex(u)

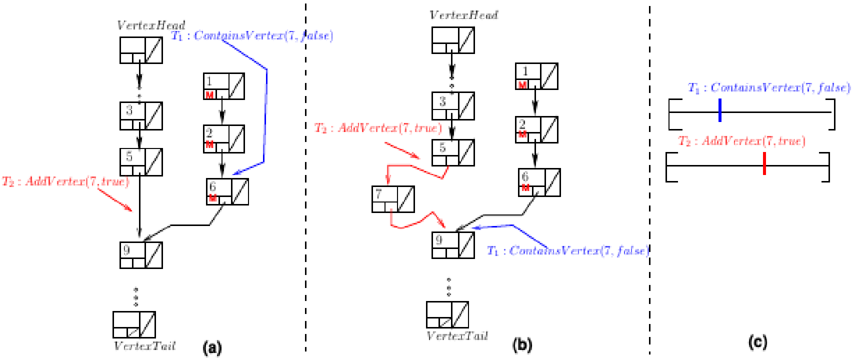
If the method returns successfully (true),

1. Point where vertex node is logically marked as deleted

If the method returns unsuccessfully,

1. Point where a vertex node with key to be deleted is not found in the vertex list

# Linearization Point of ContainsVertex(u)



# Linearization Point of AddEdge(u, v)

If the method returns successfully (true),

1. If there is no concurrent successful DeleteVertex u & v, point where new edge node is logically added or already found
2. If concurrent successful DeleteVertex(u, v), then just before its LP.

# Linearization Point of AddEdge(u, v)

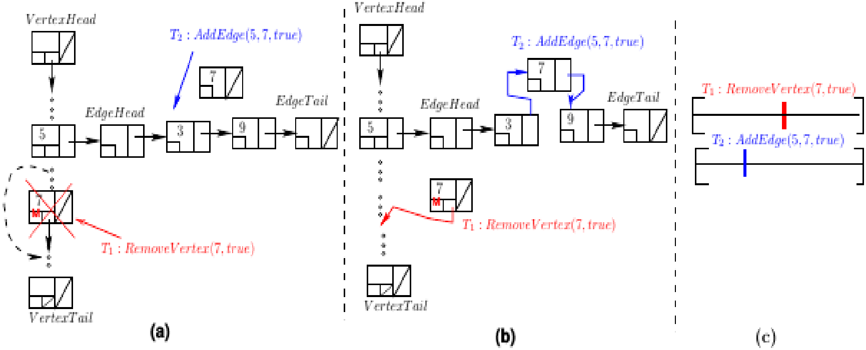
If the method returns successfully (true),

1. If there is no concurrent successful DeleteVertex u & v, point where new edge node is logically added or already found
2. If concurrent successful DeleteVertex(u, v), then just before its LP.

If the method returns unsuccessfully,

1. If there is no concurrent successful AddVertex u & v, LP is last of
    1. Point if the vertex $u$ is not found in the vertex list
    2. Point if the vertex $v$ is not found in the vertex list
    3. Point if the edge $v$ is not found in the edge list of $u$
2. If concurrent successful AddVertex u & v, then just before its LP.

# How to linearise concurrent methods?



# Linearization Point of RemoveEdge(u, v)

If the method returns successfully (true),

1. If there is no concurrent successful DeleteVertex u & v, point where new edge node is logically deleted
2. If concurrent successful DeleteVertex(u, v), then just before its LP.

# Linearization Point of RemoveEdge(u, v)

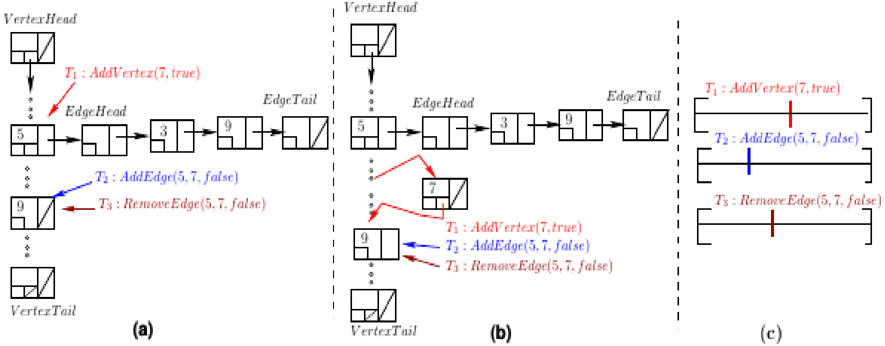
If the method returns successfully (true),

1. If there is no concurrent successful DeleteVertex u & v, point where new edge node is logically deleted
2. If concurrent successful DeleteVertex(u, v), then just before its LP.

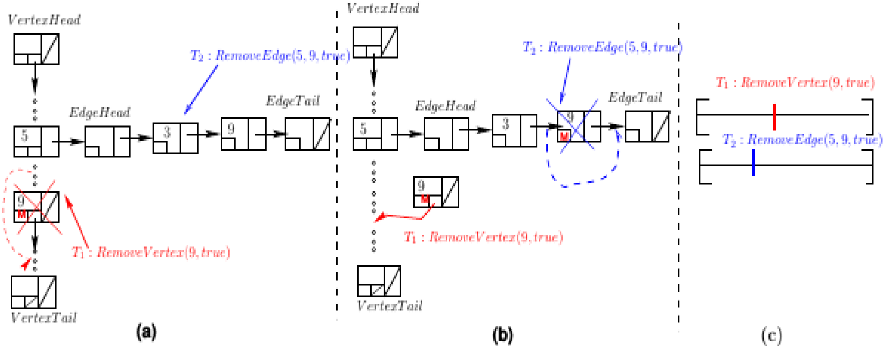
If the method returns unsuccessfully,

1. If there is no concurrent successful AddVertex u & v, LP is last of
   1. Line 9 if the vertex $u$ is not found in the vertex list
   2. Line 17 if the vertex $v$ is not found in the vertex list
   3. Line 30 if the edge $v$ is not found in the edge list of $u$
2. If concurrent successful AddVertex u & v, then just before its LP.

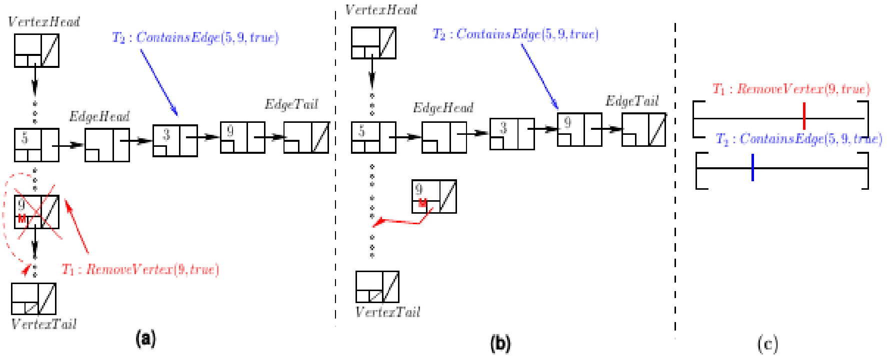
# How to linearise concurrent methods?



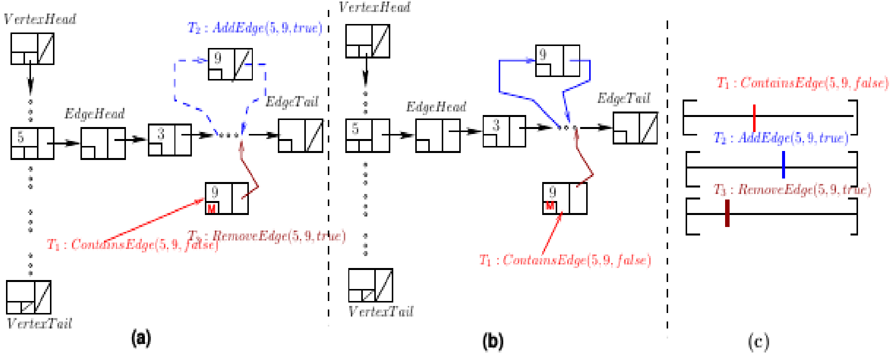
# How to linearise concurrent methods?



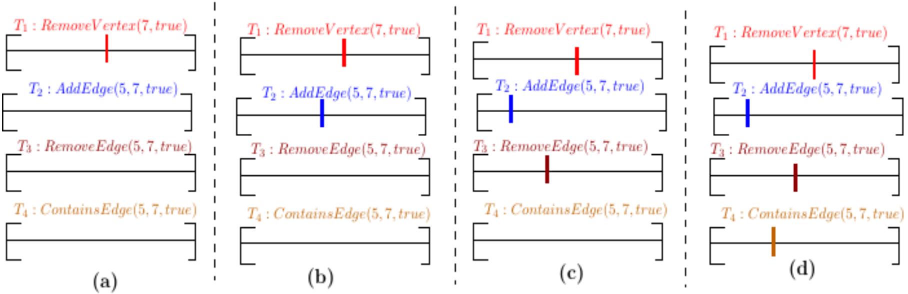
# How to linearise concurrent methods?



# How to linearise concurrent methods?



# How to linearise concurrent methods?



# Outline of the Presentation

# Experimental Setup

24 core Intel Xeon server running at 3.07 GHz core frequency

Each core supports 6 hardware threads, clocked at 1600 MHz.

Each thread randomly performs a set of operations chosen by a particular workload distribution.

Each data point is obtained after averaging for 5 iterations.

# Results 1

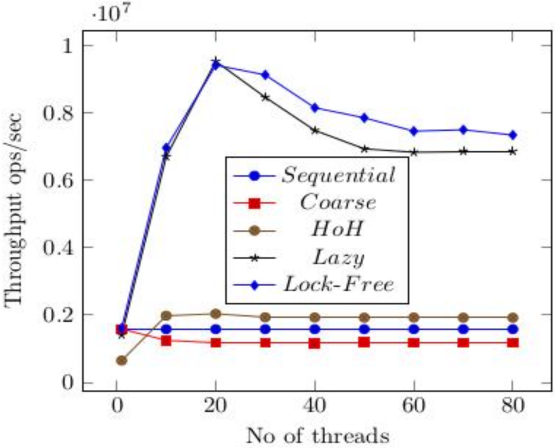



Figure : AddE:50%, DelE: 50% and rest are 0%

# Results 2

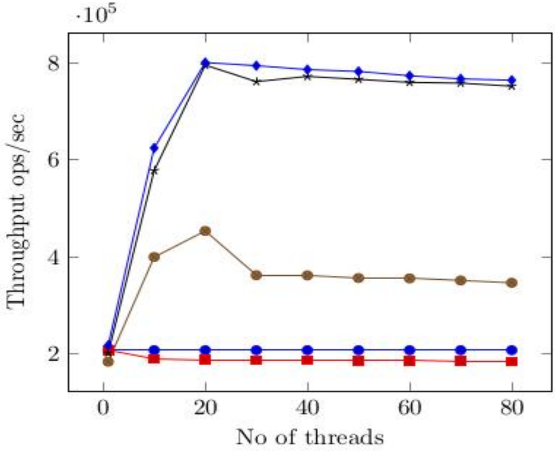



Figure : CV:15%, CE:15%, AddE:25%, DelE:10%, AddV:25% & DelV:10%.

# Results 3

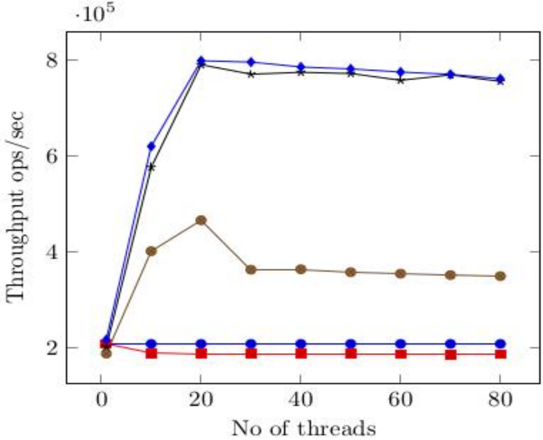



Figure : CV:40%, CE:40%, AddE:7%, DelE:3%, AddV:7% & DelV:3%

# Outline of the Presentation

# Conclusion

- Presented generic constructuction of a fully dynamic concurrent graph data structure, which allows threads to concurrently add/delete vertices/edges.

- We constructed it by the composition of the well-known concurrent list-based set data structure.

# Future Work

1. Using it for other parallel graph algorithms.
2. Currently working on Concurrent Serialization Graph Testing Scheduler.

Thank You!

# For Further Reading..


Michael A. Bender, et al. *A New Approach to Incremental Cycle Detection and Related Problems*. ACM Transactions on Algorithms, Vol. 8, No. 1, Article 3, Publication date: January 2012.

Maurice P. et al. *Linearizability: A Correctness Condition for Concurrent Objects*. ACM Transactions on Programming Languages and Systems, Vol. 12, No. 3, July 1990, Pages 463-492.

M. Herlihy. et al. *Obstruction-free synchronization: double-ended queues as an example*. Proc. IEEE ICDCS, 522-529, 2003

Y. Riany. et al. *Towards a practical snapshot algorithm*. Theoretical Computer Science, 269(1-2): 163-201, 2001.

Maurice Herlihy and Nir Shavit. *The Art of Multiprocesor Programming, Revised Print*. Imprinted Morgan Kaufmann,Elsevier, May 2012.

Peter S. Pacheco. *An Introduction to Parallel Programming, 1st Edition*. Imprinted Morgan Kaufmann,Elsevier, May 2011.

Bernhard Haeupler Telikepalli Kavitha, Roger Mathew, Siddhartha Sen, and Robert E. Tarjan, *Incremental cycle detection, toplogical ordering, and strong component maintenance*. ACM Transactions on Algorithms, 8 (2012), pp. 3:13:33

A. Natarajan and N. Mittal, *Fast concurrent lock-free binary search trees* $19^{th}$ PPoPP, 2014, pp. 317328.

E. Szpilrajn *Sur lextension de lordre partiel*. Fundamenta Mathematicae, 16 (1930), pp. 386 389

D.J. Pearce, P.H.J. Kelly, and C. Hankin, *Online cycle detection and difference propagation for pointer analysis* Proceedings of the 3rd International Workshop on Source Code Analysis and Manipulation, Sept. 2003, pp. 312

Donald E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms* ,Addison Wesley, Reading, MA, 2nd ed., 1973.

A.B. Kahn, *Topological sorting of large networks*, Communications of the ACM, 5 (1962),pp. 558562.

Questions?