Wait-Free Updates and Range Search using Uruv

Gaurav Bhardwaj, Bapi Chatterjee, Abhay Jain, **Sathya Peri**

SSS 2023

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Table of Contents

B+Tree

- B+ tree is a balanced tree data structure that maintains sorted data and allows searches, insertions, deletions, and sequential access in logarithmic time.
- Types of Nodes:
  - Internal Nodes.
  - Leaf Nodes.
- In an m-ordered B+Tree each node contains between [m/2] - 1 to m - 1 keys except root.
- Similarly each internal node must contains at-least [m/2] children.
- All the keys must be in the ascending order.
- All actual keys are stored at the leaf nodes.

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

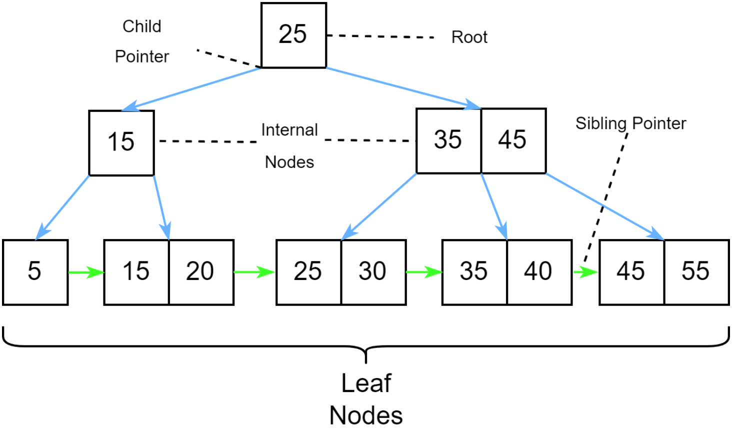


Figure: B+Tree

Insertion and Split

- Insertion finds the leaf node where a key should be located, and splits full nodes as necessary to allow new elements to be inserted.
- Split- Starts at a full leaf node where a new element should be placed, and propagates up the tree.

Insertion and Split

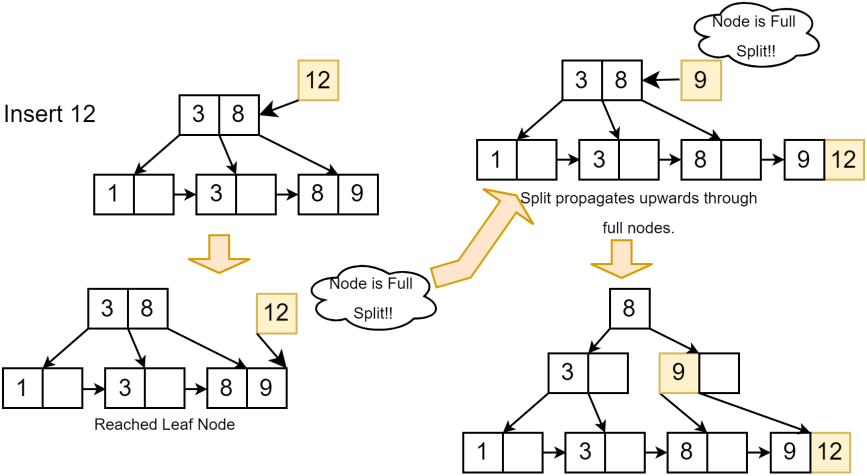


Figure: Insertion and Split

Deletion

- Similar to insertion but bit more complicated.
- Instead of split merge is performed in order to balance the tree.

Deletion

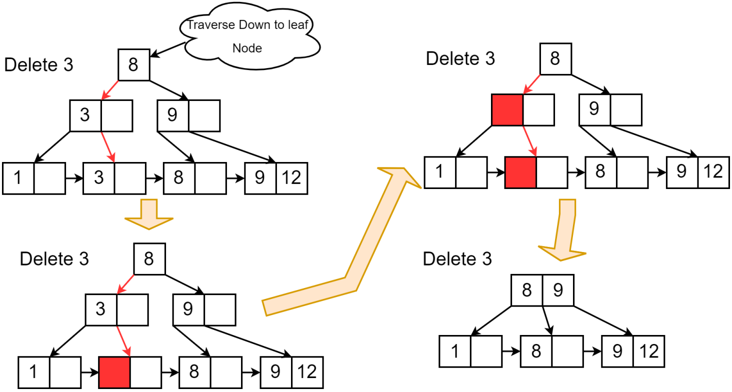


Figure: Deletion and Merge

B+Tree (Benefits)

- Efficient Searching and Retrieval.
- Range Queries.
- Sequential Access.
- Easy to Balance.
- Stable Performance.
- Widely Used in Databases.

Concurrency

- In modern applications, the ability to handle concurrent operations is crucial, especially in databases processing large volumes of data simultaneously.
- Concurrent B+Trees are essential for real-time systems where data consistency and responsiveness are paramount.
- Without proper synchronization, concurrent access can lead to race conditions, jeopardizing data integrity.
- Inadequate concurrency control might cause deadlocks, halting system operations and causing delays.

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

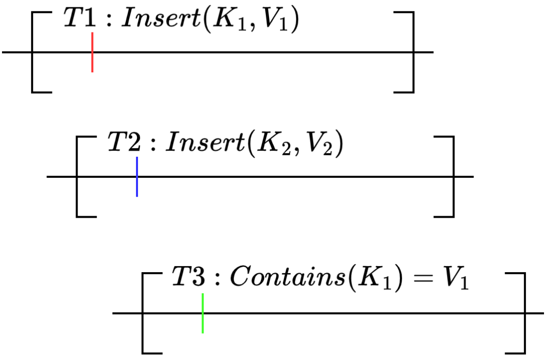


Figure: Linearizability Example

Progress Condition: Avoiding Locks

- Deadlock Avoidance is hard.
- A low-priority process may hold the lock on a resource desired by a high-priority process.
- Only few percentage of process can access the critical section at a time hampers the performance.

Non-Blocking Progress Condition

An execution is said to be Non-Blocking if it doesn't blocks the execution of other threads.

- **Obstruction Free:** A thread is guaranteed to finish in a finite number of steps in isolation.
- **Lock-Freedom:** Atleast one thread should be able to finish in the finite number of steps.
- **Wait-Freedom:** All threads should be able to finish in a finite number of steps.

Problem Statement

Develop a state of the art Wait-Free Concurrent B+Tree based datastructure which supports consistent Range Queries.

ADT operations

- An $\textsc{Insert}(K, V)$ inserts the key K and an associated value V if $K \notin \mathcal{K}$.
- A $\textsc{Delete}(K)$ deletes the key K and its associated value if $K \in \mathcal{K}$.
- A $\textsc{Search}(K)$ returns the associated value of key K if $K \in \mathcal{K}$; otherwise, it returns -1. It does not modify $(\mathcal{K}, \mathcal{V})$.
- A $\textsc{RangeQuery}(K_1, K_2)$ returns keys $\{K \in \mathcal{K} : K_1 \leq K \leq K_2\}$, and associated values without modifying $(\mathcal{K}, \mathcal{V})$; if no such key exists, it returns -1.

Proactive Approach:

- If the number of keys in a node exceeds/falls short of its maximum/minimum threshold after an insertion/deletion, it requires splitting/merging.
- Splitting/Merging may cascade to the root.
- **Proactive Approach:** checks threshold of nodes whiles traversing down a tree every time; if a node is found to have reached its threshold, without waiting for its children, a preemptive split or merge is performed.

Table of Contents

Related Work

- **Lock-Free B+Tree**[a]: Each node implements a linked list augmented by an array.
- **OpenBWtree**[b] an optimized lock-free B+tree that was designed to achieve high performance under realistic workloads.'
- **Bundeled Reference**[c]: A lock-based approach using versions for wait-free range search.
- **Constant time snapshot**[d]: A lock-free approach using versions for wait-free range search.

---

[a] Anastasia Braginsky and Erez Petrank (2012). "A lock-free B+ tree". In: *Proceedings of SPAA*, pp. 58–67.

[b] Ziqi Wang et al. (2018). "Building a bw-tree takes more than just buzz words". In: *Proceedings of the 2018 International Conference on Management of Data*, pp. 473–488.

[c] Jacob Nelson, Ahmed Hassan, and Roberto Palmieri (2021). "Bundled references: an abstraction for highly-concurrent linearizable range queries". In: *PPOPP 2021*, pp. 448–450.

[d] Yuanhao Wei et al. (2021). "Constant-time snapshots with applications to concurrent data structures". In: *PPOPP 2021*, pp. 31–46.

Drawback

- No existing approach supports proactive ADT operations, which leads to the cascading effect. This cascading effect leads to the performance hamper in concurrent settings.
- No existing tree-based structure supports the consistent range search.
- No wait-free tree-based data structure exists in the literature.

Our Contributions

- We have developed a Wait-Free Concurrent B+Tree based data structure using Proactive approach.
- Our implementation supports wait-free consistent ranngequery.

Table of Contents

URUV Datastructure Design

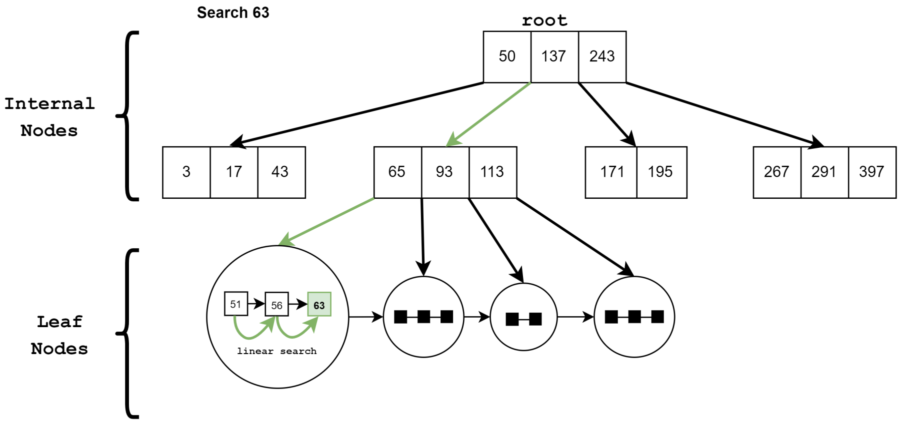


Figure: Example of Uruv's design. In this example, a search operation is being performed wherein the green arrows indicate a traversal down Uruv, and we find the key, highlighted green, in the linked-list via a linear search.

URUV Datastructure Design

- We are using versioned linked list at leaf nodes for the consistent rangesearch.
- For each node in the linked list there is pointer to the version list.
- each node (vnode) in the version list contains the value for the corresponding key aand the timestamp (ts).
- Each item the value for the corresponding key changes is been updated on the head of the version list along with the timestamp.

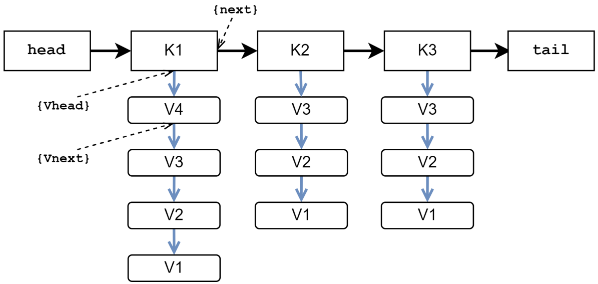


Figure: Versioned Linked List

URUV Object Structure

```
Vnode{
    value_t value;
    int ts;
    Vnode* nextv;
}

llNode{
    key_t key;
    Vnode* vhead;
    llNode* next;
}

VLF_LL{

    llNode* head;
}
```

Figure: Versioned Lock-Free Linked-List Data Structure

```
Uruv{
    int global_ts;
    Node* root;
}
InternalNode: Node{
    long key[MAX]
    Node* ptr[MAX+1]
    helpidx
}

LeafNode: Node{
    VLF_LL* ver_head;
    LeafNode* next;
    LeafNode* newNext;
    int ts
}

Node{
    long count;
    bool isLeaf;
    bool frozen;
}
```

Figure: URUV detailed structure

Tree Insert(126)

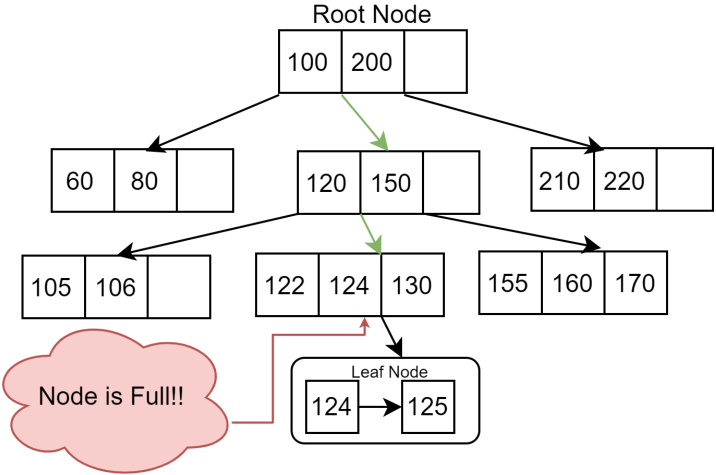


Figure: URUV Tree Traversal

Freeze Internal

- Internal Node is frozen by marking the each child pointer one by one.

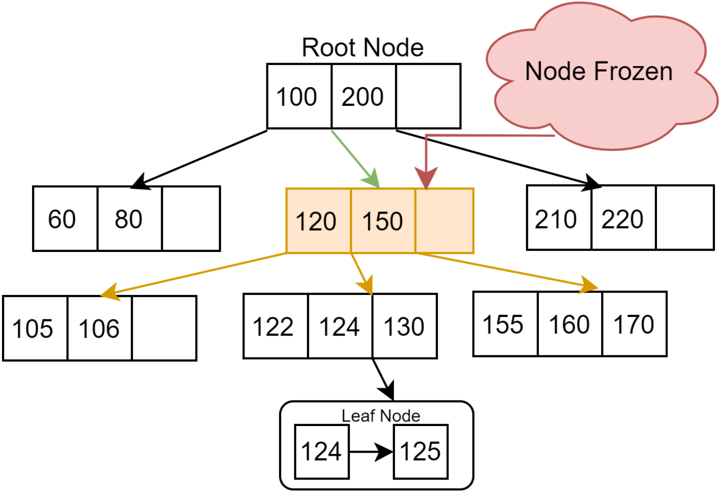


Figure: Freezing Parent Node

Freeze Internal

- Once a node is frozen then no thread can do changes to that node.
- If any thread finds any node frozen or undergoing freezing then that thread helps in freezing and performing split/merge operation.

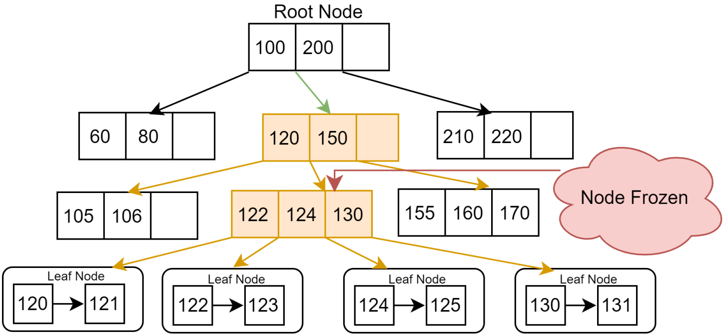


Figure: Freezing Leaf Node

Split Internal

- Once the node to be split and its parents are frozen we can perform the split operation.
- We create the new parent node and the splitted node and replace with the current pointer using **CAS**.

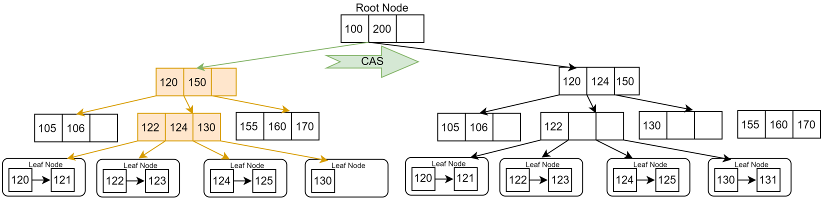


Figure: Splitting Internal Node

Split Leaf

- After traversing down to leaf node if the leaf node is full we split the leaf node.

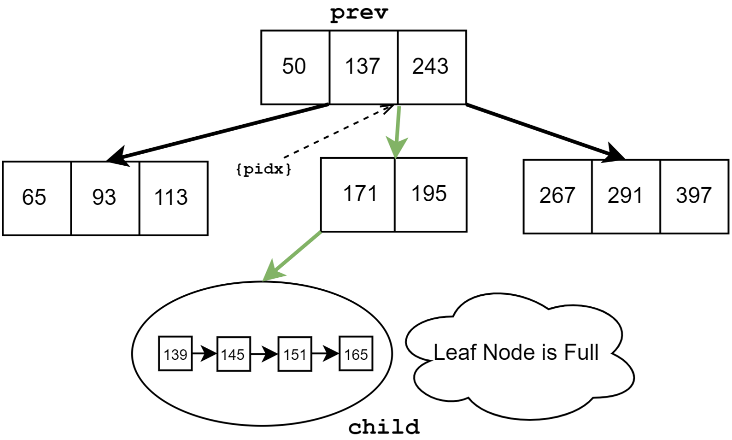


Figure: Split Leaf

- Just like internal node parent of the leaf node to be splitted is frozen.

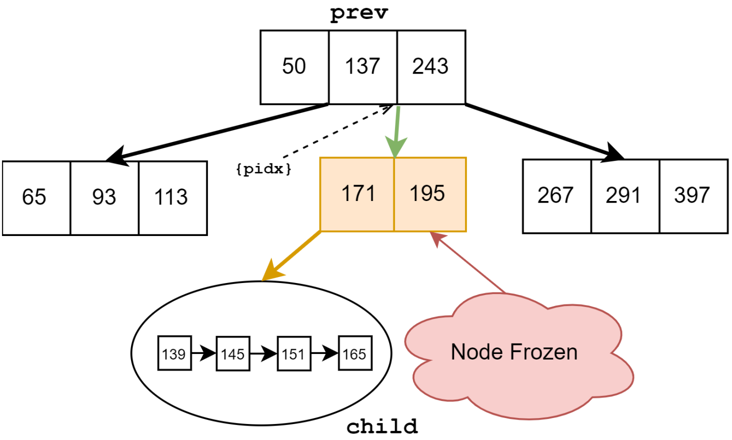


Figure: Split Leaf

Split Leaf

- After freezing the parent node leaf node is also frozen.
- Leaf node is frozen by marking the next pointer of the linked list node and the head of the version list.

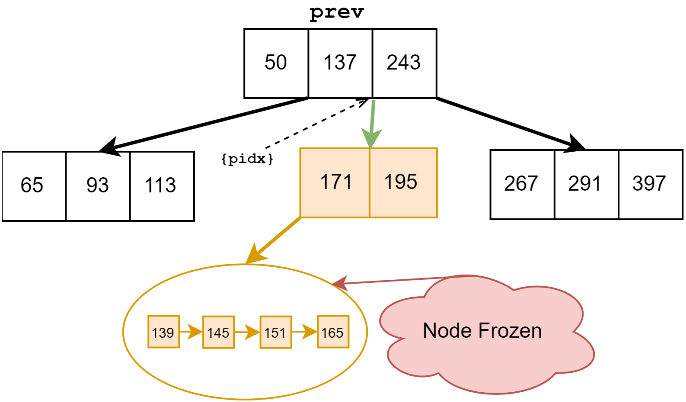


Figure: Split Leaf

Split Leaf

- After freezing the leaf node it is splitted into two parts.
- After splitting the leaf node a new parent node is created containing the new splitted nodes are replaced with the current one using **CAS**

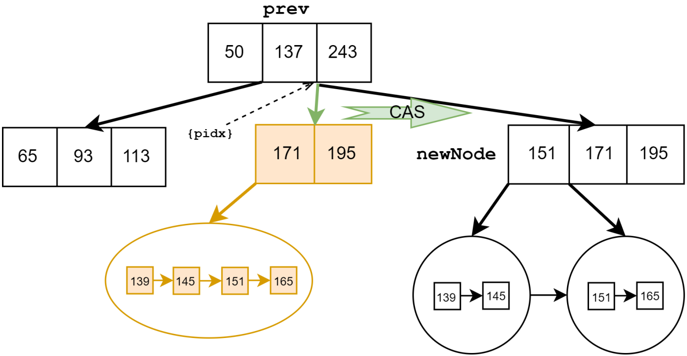


Figure: Split Leaf

Merge Leaf

- Merge Leaf is similar to the split leaf where we merge two leaf nodes instead of split.

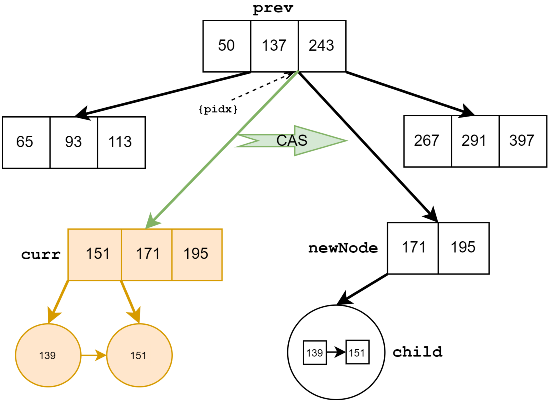
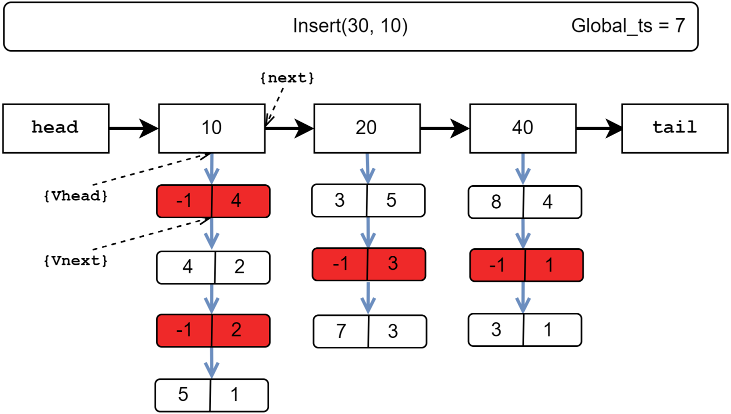


Figure: Merge Leaf

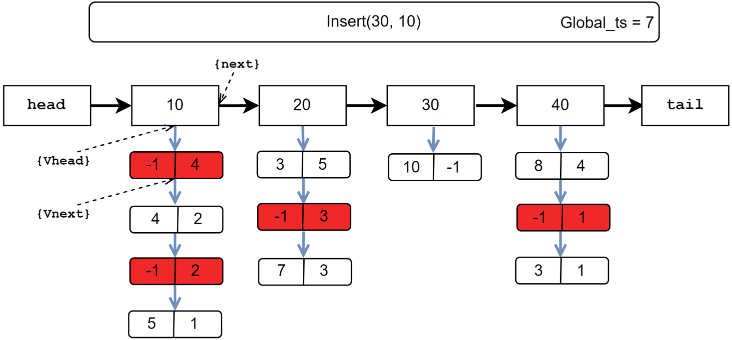
Insert Leaf

- After Traversing down to the leaf node we add the key to the linked list if it is not present.

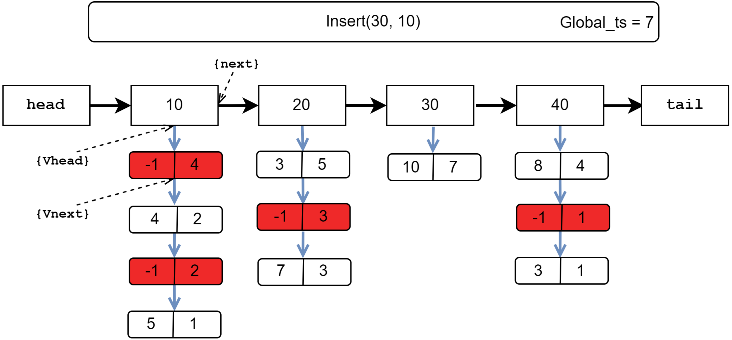


Insert Leaf

- Intially after inserting the node to the linked list ts of vnode at vhead is -1.
- which is later initialise with the global_ts.

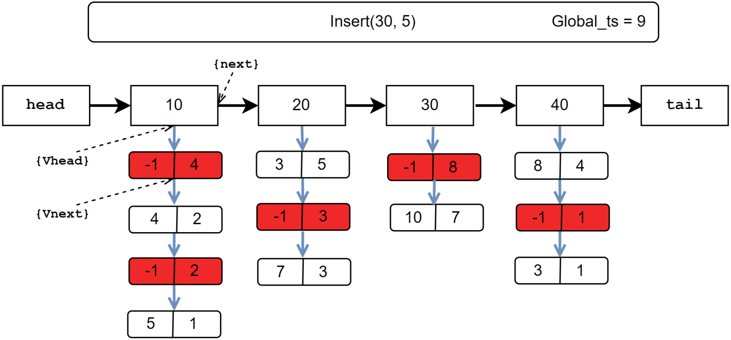


Insert Leaf

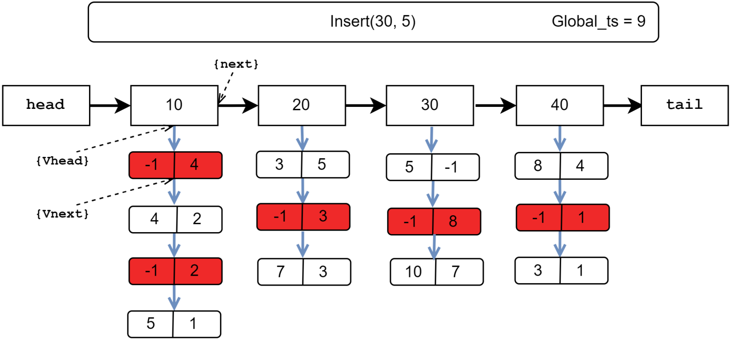


Insert(30, 10)
Global_ts = 7
{next}
head
10
20
30
40
tail
{Vhead}
{Vnext}
-1 4
4 2
-1 2
5 1
3 5
-1 3
7 3
10 7
8 4
-1 1
3 1

- If the key is already there in the linked list we will add a new version node on the vhead of the linked list node with the current timestamp.

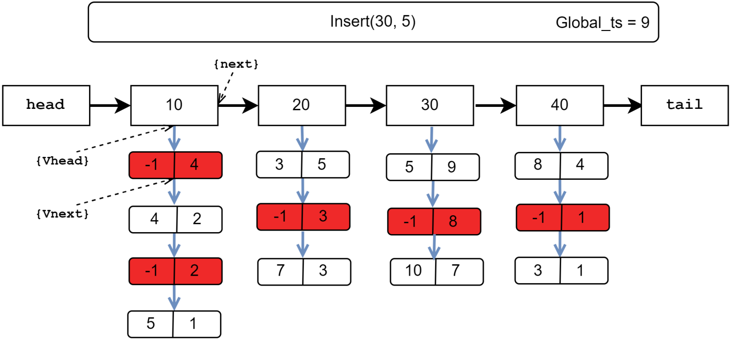


Insert Leaf



Insert(30, 5)
Global_ts = 9
{next}
head
10
20
30
40
tail
{Vhead}
{Vnext}
-1 4
4 2
-1 2
5 1
3 5
-1 3
7 3
5 -1
-1 8
10 7
8 4
-1 1
3 1

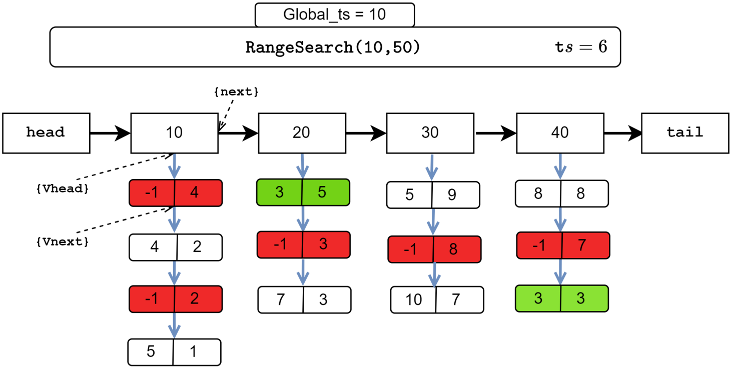
Insert Leaf



Insert(30, 5)
Global_ts = 9
{next}
head
10
20
30
40
tail
{Vhead}
{Vnext}
-1 4
4 2
-1 2
5 1
3 5
-1 3
7 3
5 9
-1 8
10 7
8 4
-1 1
3 1

Range Search

- Rangesearch operation fetch the global_ts and increment it by 1.
- It is linearized when it fetches the global_ts.
- It collects all the methods whose timestamp is lower or equal them the timestamp of RangeSearch.



Wait Freedom

- Wait-freedom is achieved using fast-path-slow-path method[e].
- Wait-free operation starts exactly as the lock-free algorithm. [Fast-Path]
- If a thread cannot complete its operation even after several attempts, it enters the slow path by announcing that it would need help.
- Global stateArray is maintained to keep track of the operations that every thread currently needs help with.
- In the slow path, an operation first publishes a State object containing all the information required to help complete its operation.

[e] Shahar Timnat et al. (2012). "Wait-Free Linked-Lists". In: *OPODIS*, pp. 330–344.

Wait-Freedom

```
State* stateArray[totalThreads]

class HelpRecord{
    long currTid;
    long lastPhase;
    long nextCheck;
}
```

```
class State{
    long phase;
    bool finished;
    Vnode* vnode;
    long key;
    long value;
    llNode* searchNode
}
```

Figure: Data structures used in wait-free helping

Table of Contents

Results

- We have compared the lock-free as well as the wait-free version of our implementation (URUV) with its counter parts such as **LF_B+Tree**[f] and **Open_BwTree**[g] for the update operation.
- For Rangesearch we have compared URUV with **VCAS-BST**[h].

[f] Anastasia Braginsky and Erez Petrank (2012). "A lock-free B+ tree". In: *Proceedings of SPAA*, pp. 58–67.

[g] Ziqi Wang et al. (2018). "Building a bw-tree takes more than just buzz words". In: *Proceedings of the 2018 International Conference on Management of Data*, pp. 473–488.

[h] Yuanhao Wei et al. (2021). "Constant-time snapshots with applications to concurrent data structures". In: *PPOPP 2021*, pp. 31–46.

Results

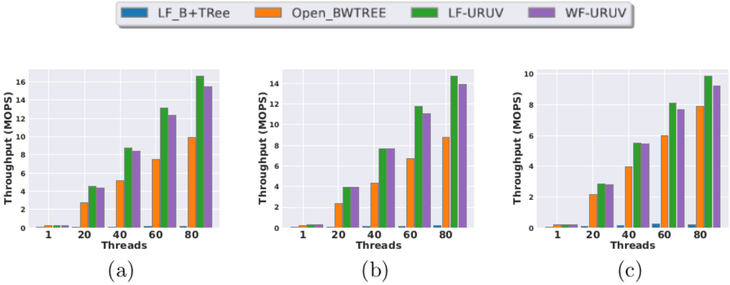


Figure: The performance of **Uruv** when compared to **LF_B+Tree** and **Open_BwTree**. Higher is better. The workload distributions are (a) Reads - 100% (b) Reads - 95%, Updates - 5%, and (c) Reads - 50%, Updates - 50%

Results

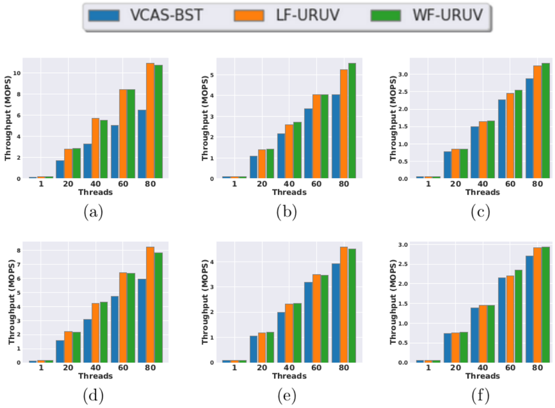


Figure: The performance of **Uruv** when compared to **VCAS-BST**. The workload distributions are (a) Reads - 94%, Updates - 5%, Range Queries of size 1K - 1%, (b) Reads - 90%, Updates - 5%, Range Queries of size 1K - 5%, (c) Reads - 85%, Updates - 5%, Range Queries of size 1K - 10%, (d) Reads - 49%, Updates - 50%, Range Queries of size 1K - 1%, (e) Reads - 45%, Updates - 50%, Range Queries of size 1K - 5%, and (f) Reads - 40%, Updates - 50%, Range Queries of size 1K - 10%

Table of Contents

Conclusion

- **URUV** is the first wait-free tree-based concurrent data structure.
- **URUV** proposed the first concurrent wait-free solution for proactive self-balancing trees.
- **URUV** is the fastest wait-free tree-based data structure with consistent range queries.

Questions??