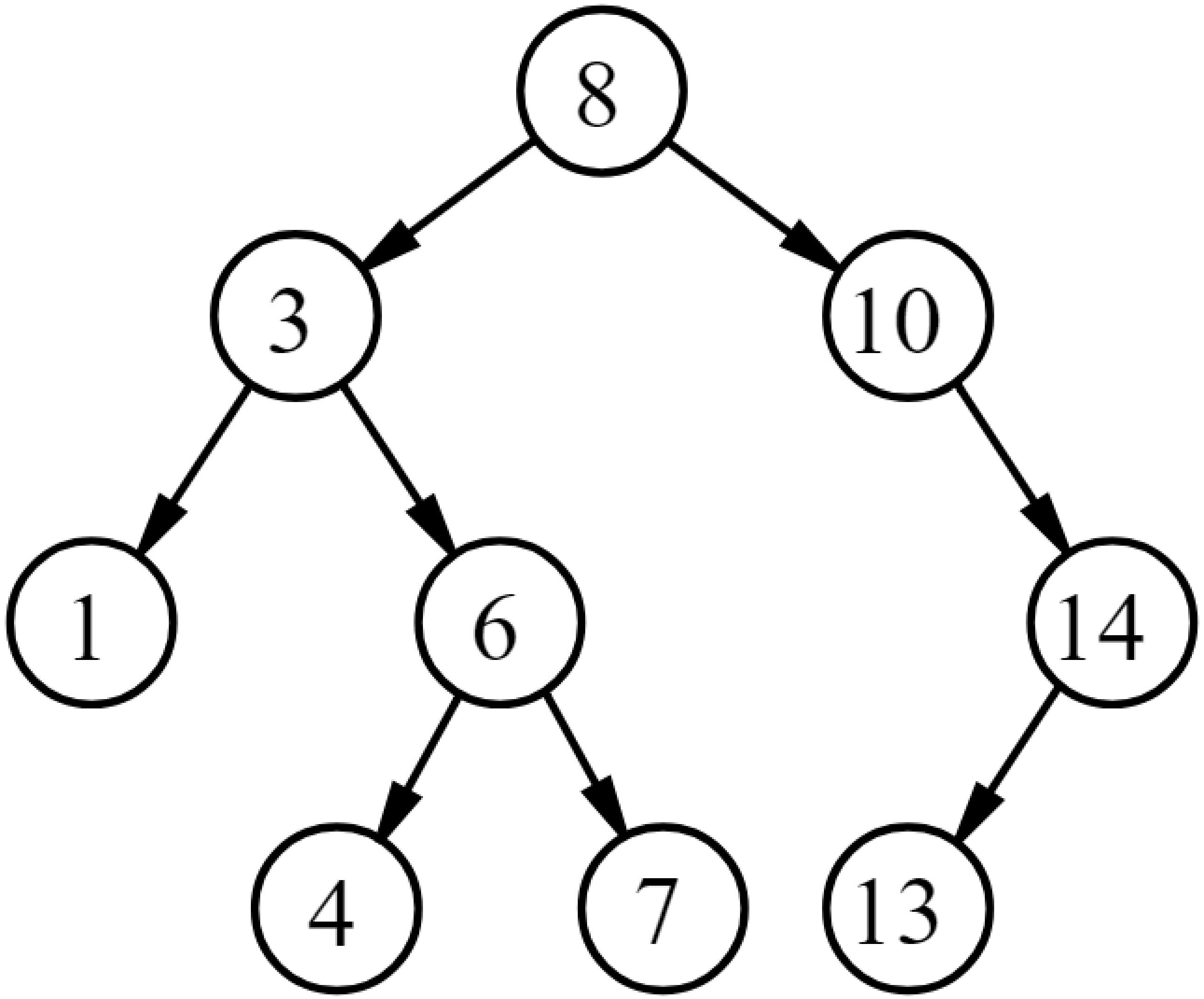
Lock Free Binary Search Trees

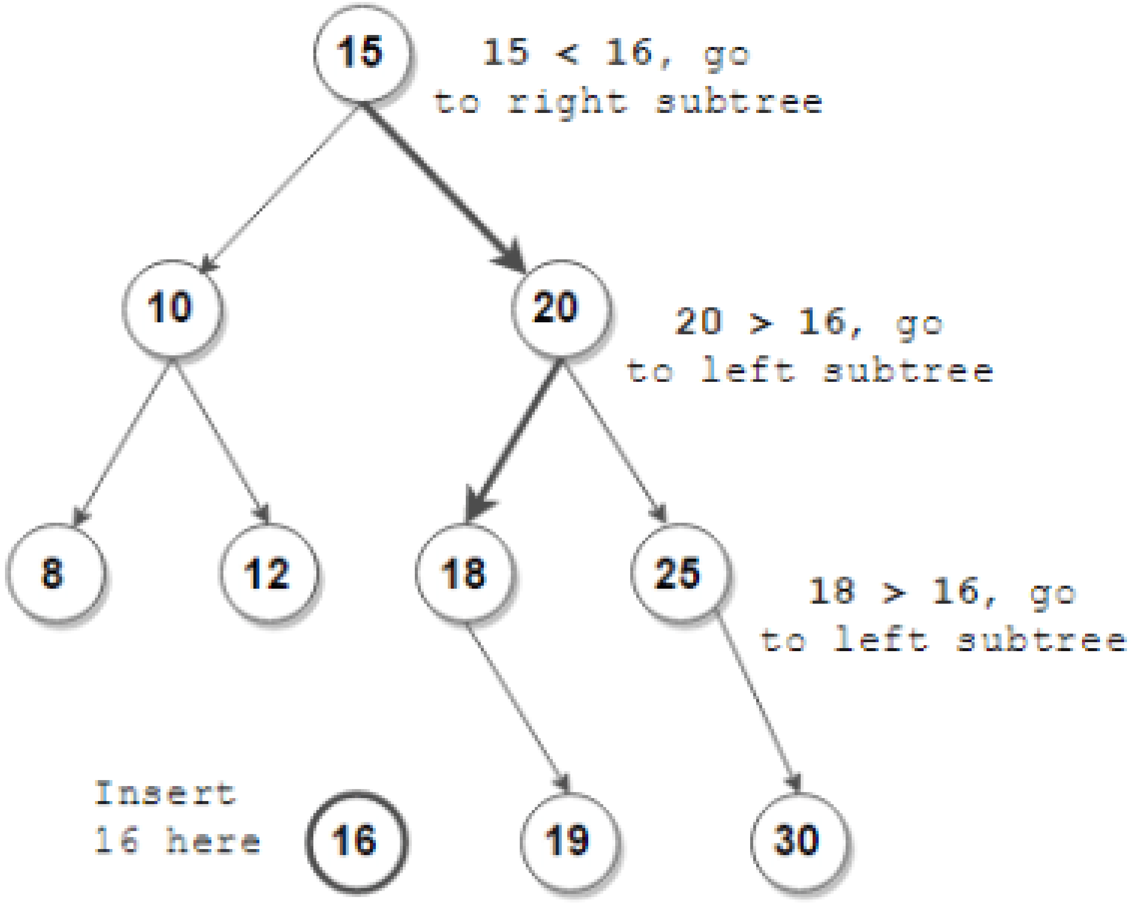
S.M. Iqbal Morshed
Fnu Tulha

Binary Search tree

- A Binary Search Tree contains keys that are taken from an ordered universe.
- Build on the concept of binary search.

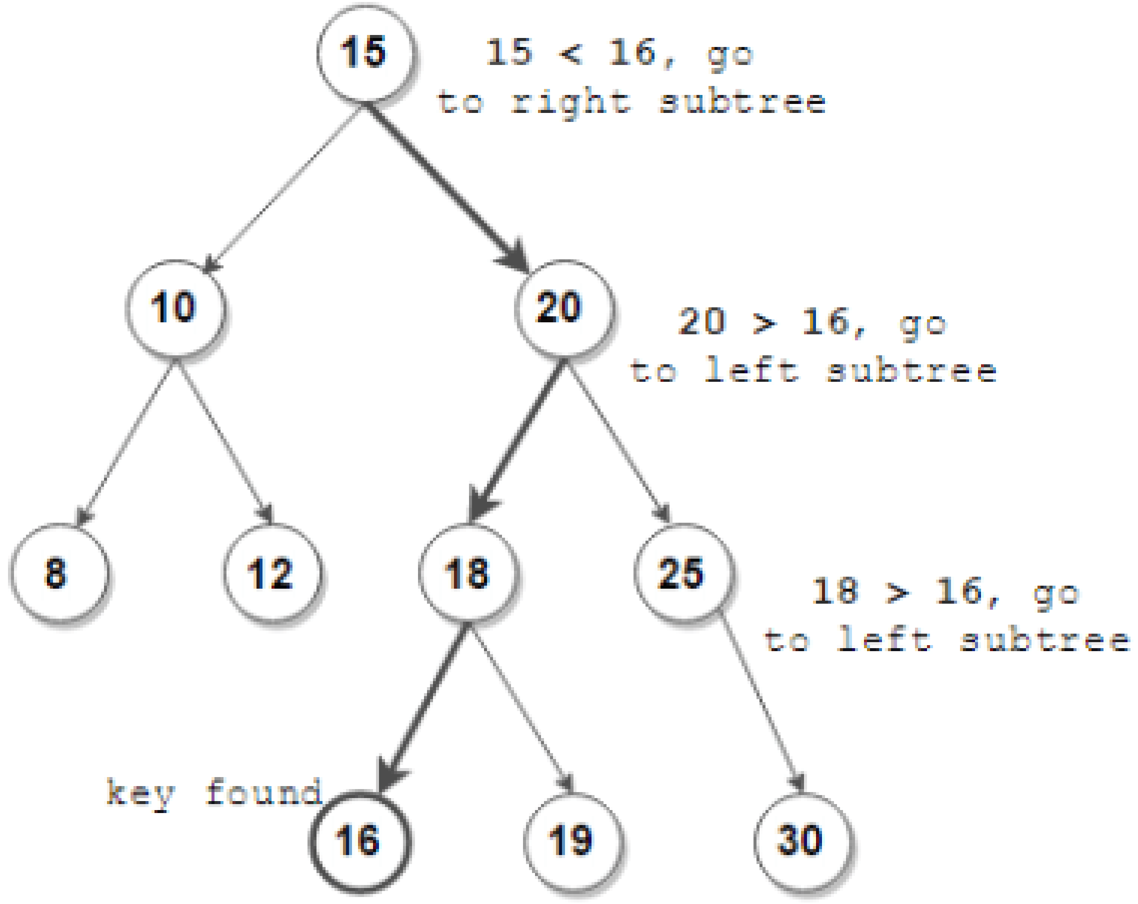


BST Insertion



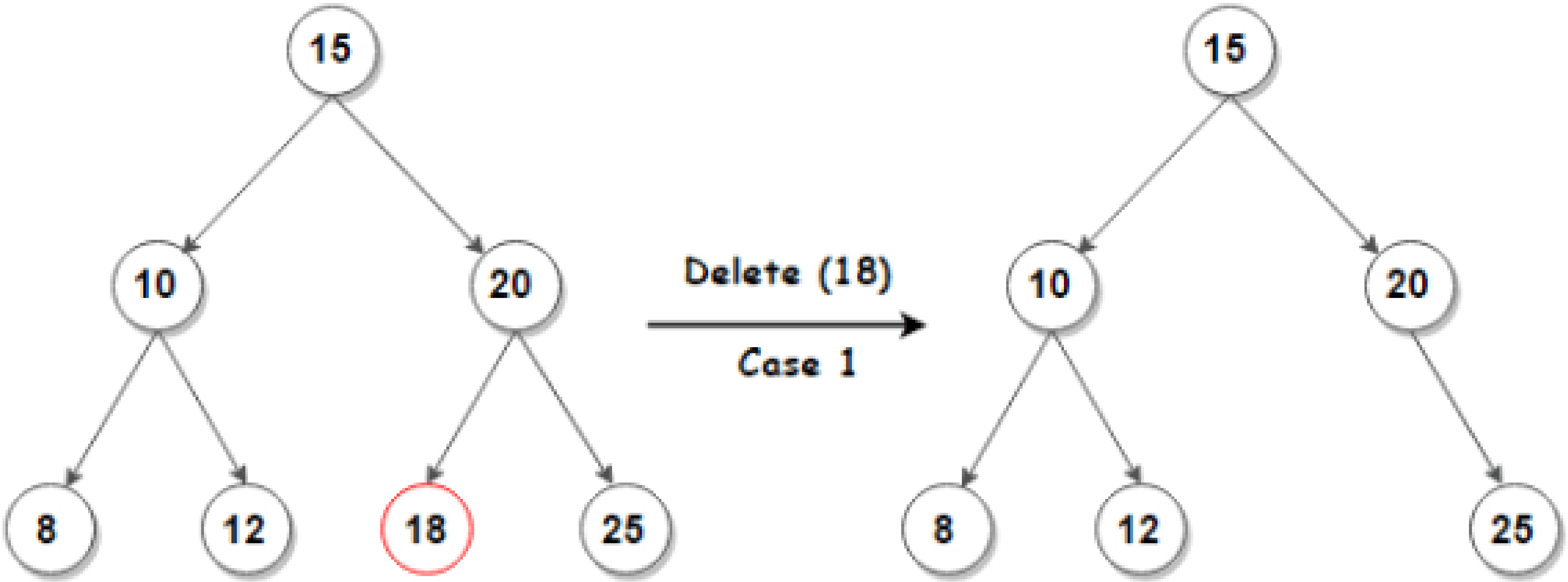
Insert (root, 16)

BST Search

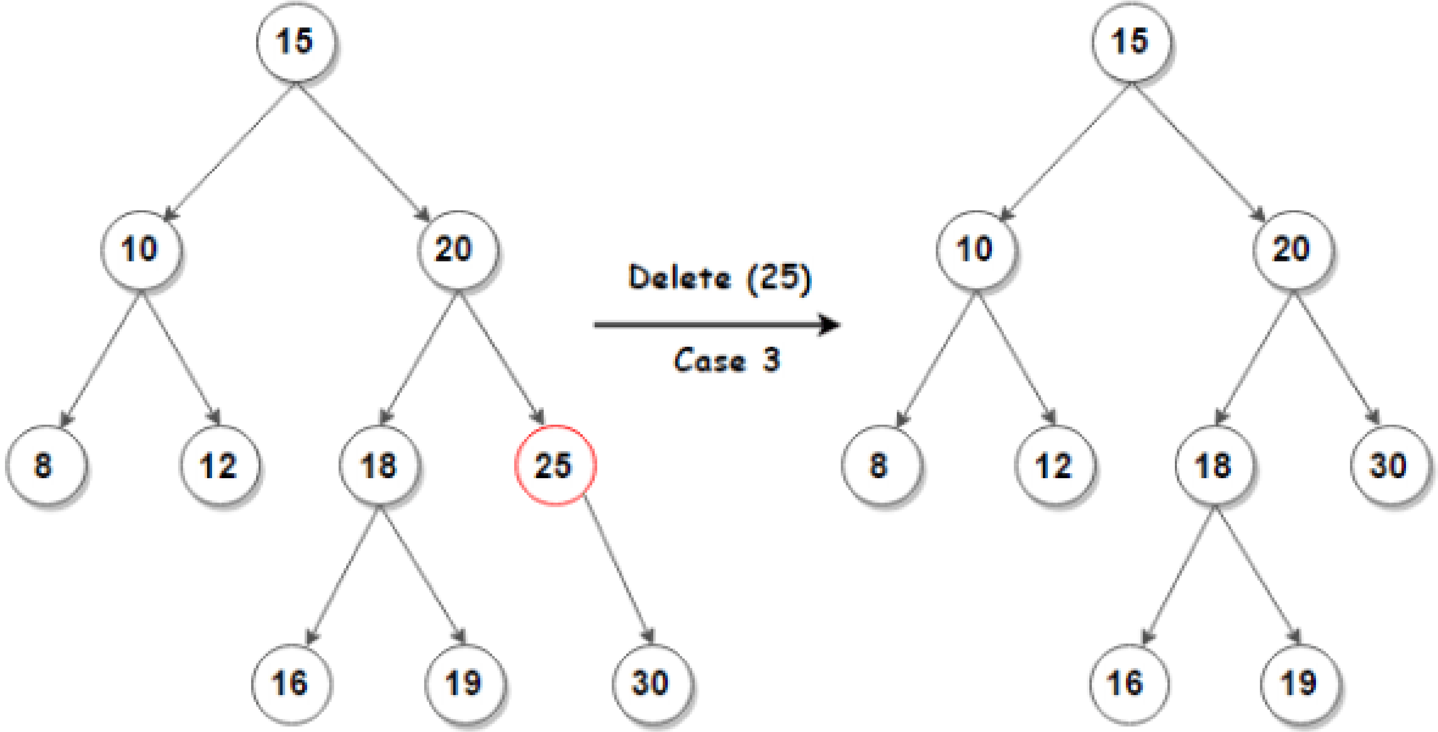


Search (root, 16)

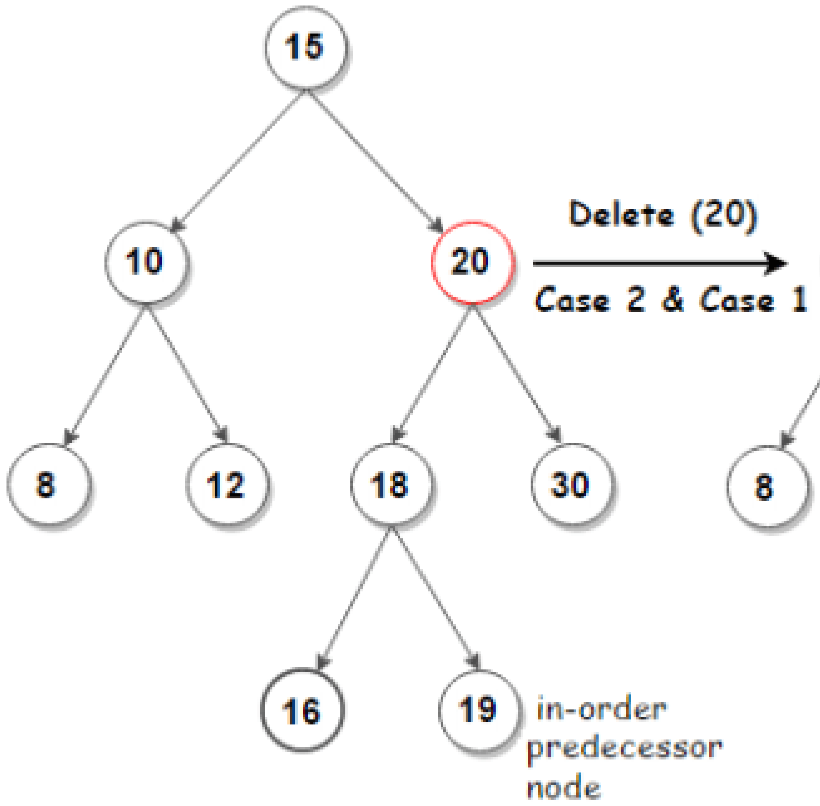
BST Deletion
Case 1
No Children



BST Deletion
Case 2
One Child



BST Deletion
Case 3
Two Children

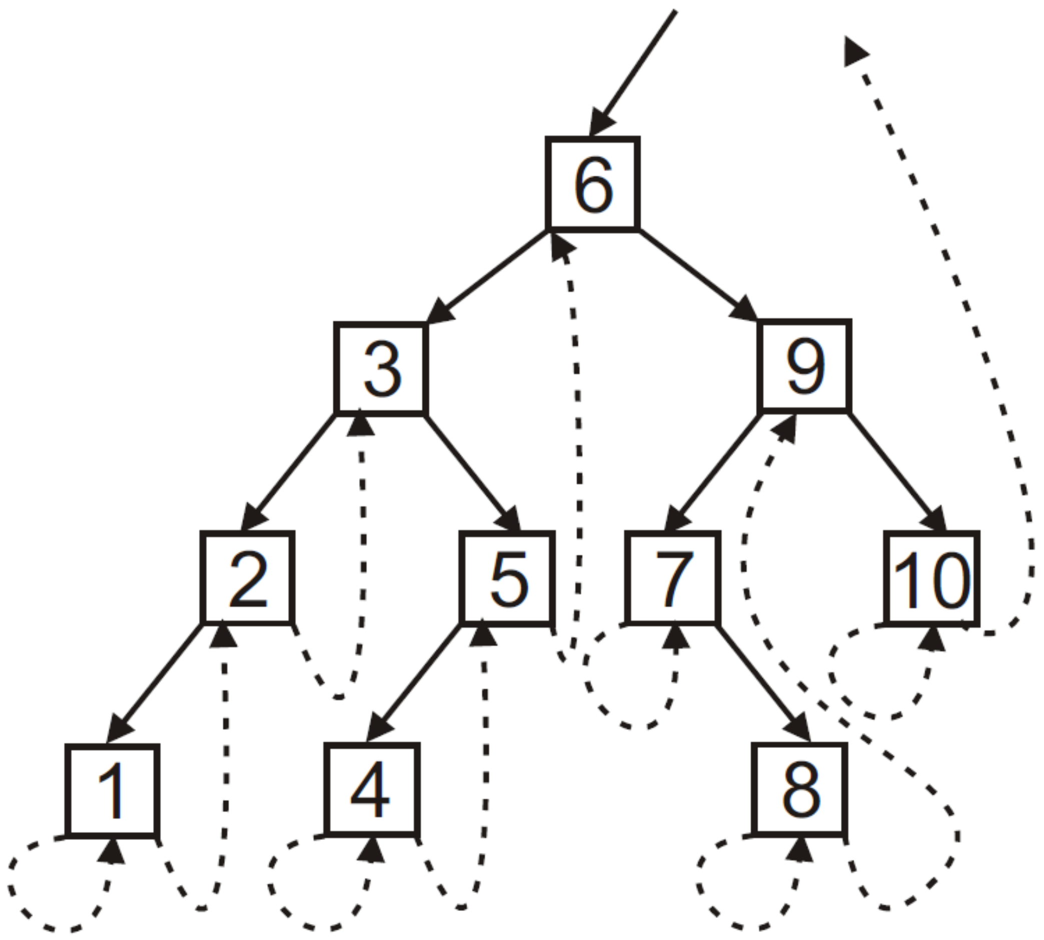


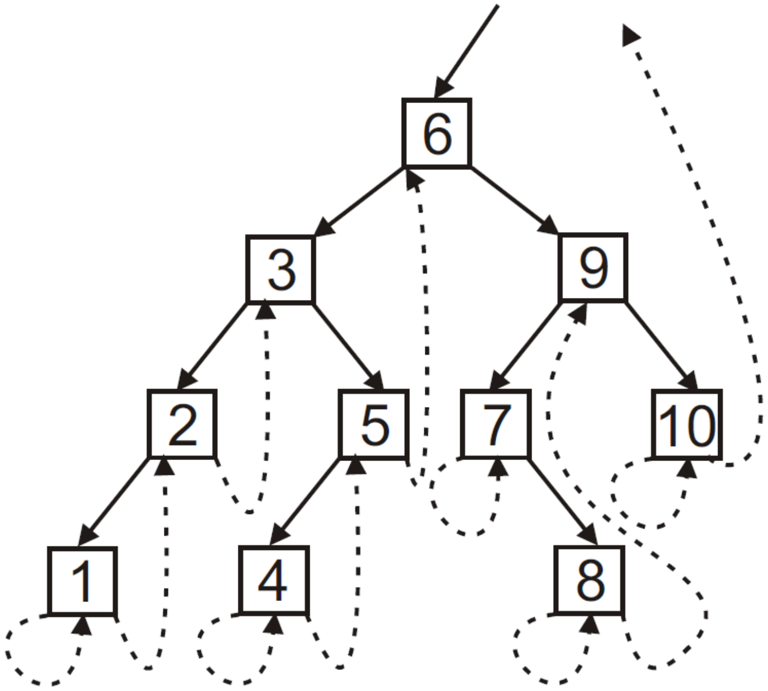
Lock Free BST

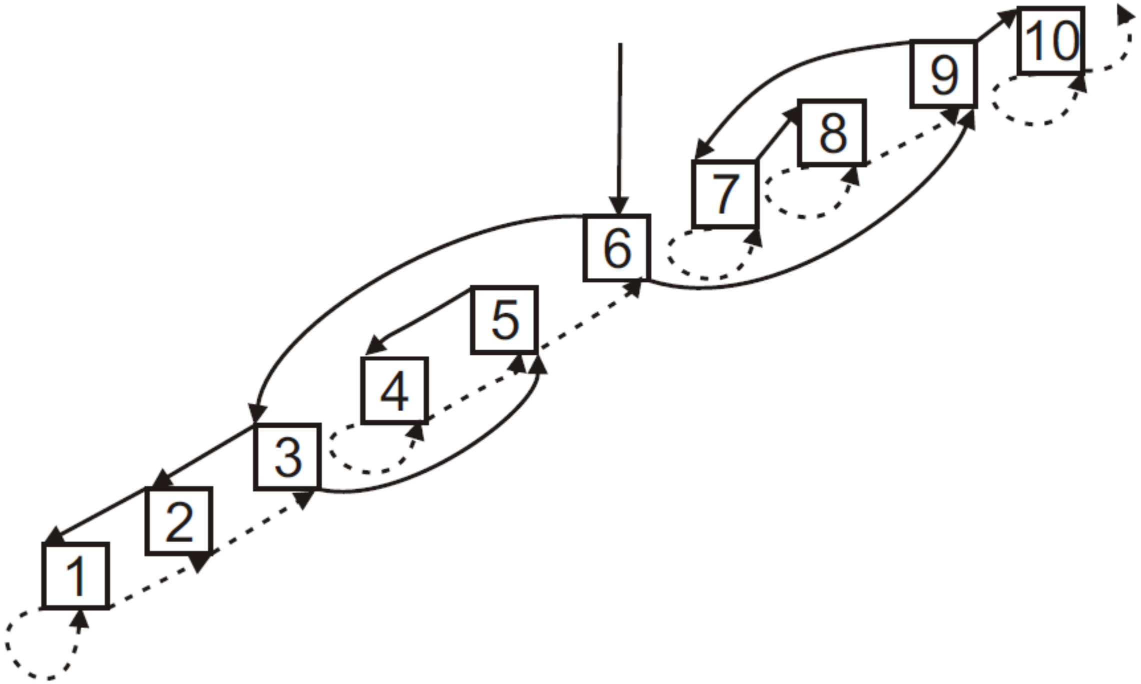
- Problems in concurrent BST
  - Multiple threads contend for deleting leaf node -> unsuccessful threads restart from the root.
  - Time Complexity: O(c*h(n))
    - where c = number of contending threads, h = height of tree for n nodes
  - "Contains" needs to be aware of concurrent remove of binary node.
    - Leads to invalid state otherwise.

Threaded BST

- Use of threaded links according to the approach in [3].
- If leaf node, threaded link to itself and successor.
- If unary node, threaded link to itself OR successor.
- If binary node, an outgoing threaded link does not exist.



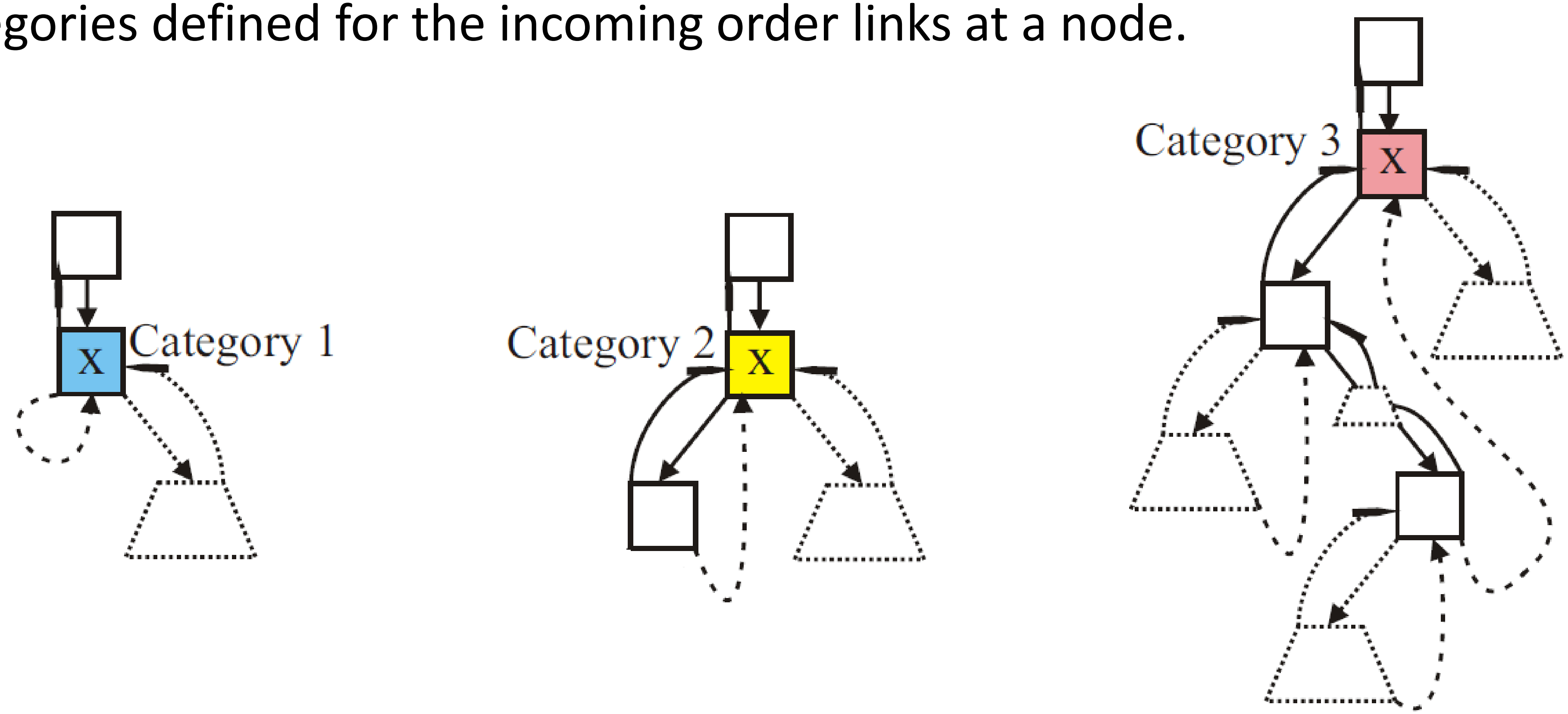




Why a Threaded BST?

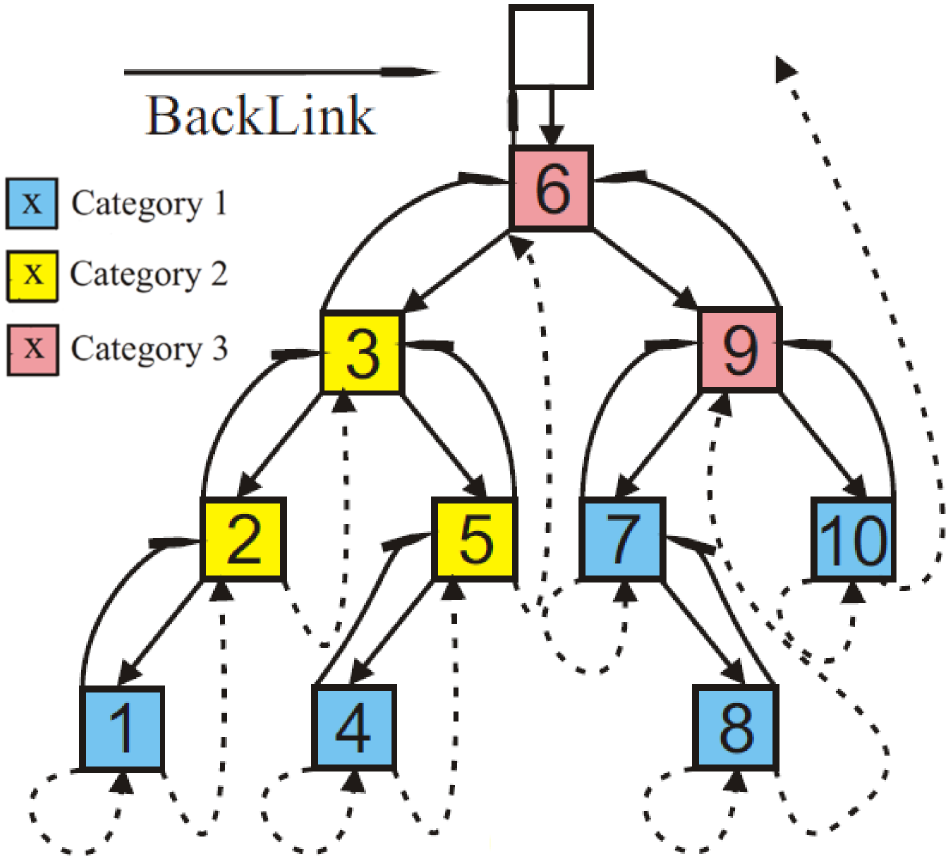
Other Features

- An incoming threaded link is an order link.
- Categories defined for the incoming order links at a node.



Updated BST

- Backlinks from node to parent.
- prev and succ obtained according to in order traversal.
- Mark, Flag and Thread bit used for each link.
- Each link of a node is represented by:
  - (Reference, flag, mark, thread)
- Important: CAS takes this argument as well.
  - For example, (R, 0, 0, 1)



Set ADT

- Supports add, contains and remove.
- Add(k) -> Add node with key k to a BST.
- Contains(k) -> return true if a node with key k is found.
- Remove(k) -> remove node with key k.

Contains Operation

- Contains can help marked nodes.
- Condition: Starting from curr and prev, we traverse the BST:

While(true):

if k_curr == k:

return 2;

If(help):

help(marked)

else if link == leftThreaded:

stop; // **since the key is not present**

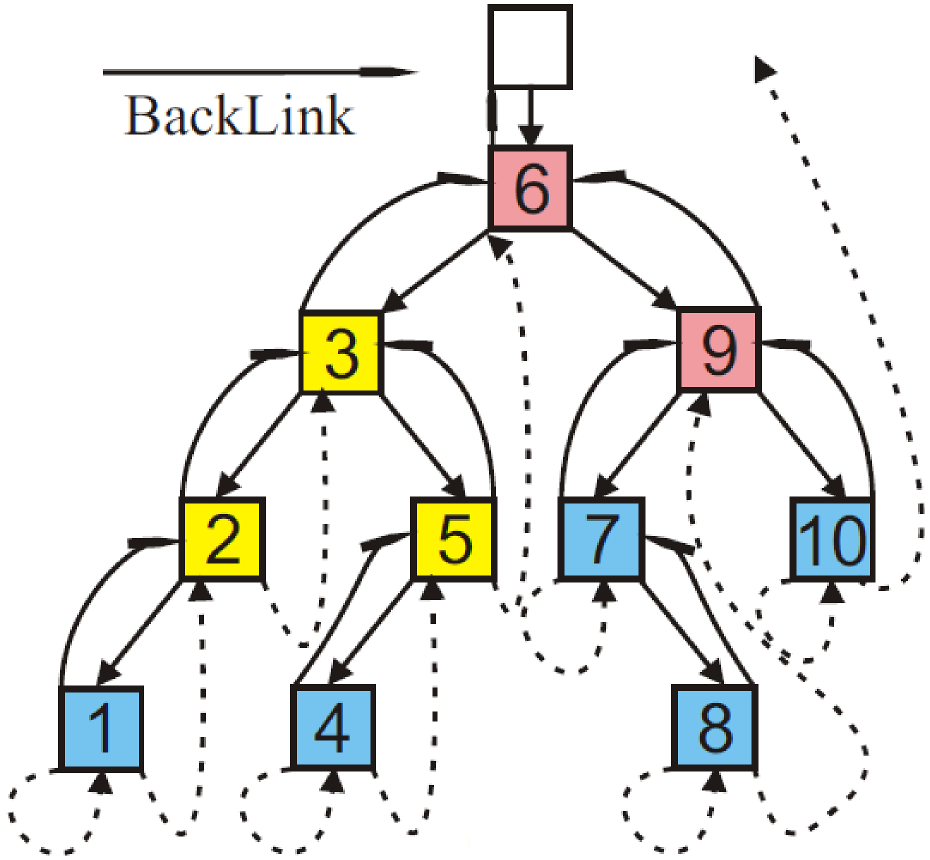
else if link==rightThreaded:

if k < k_next:

stop; **since the key is not present**

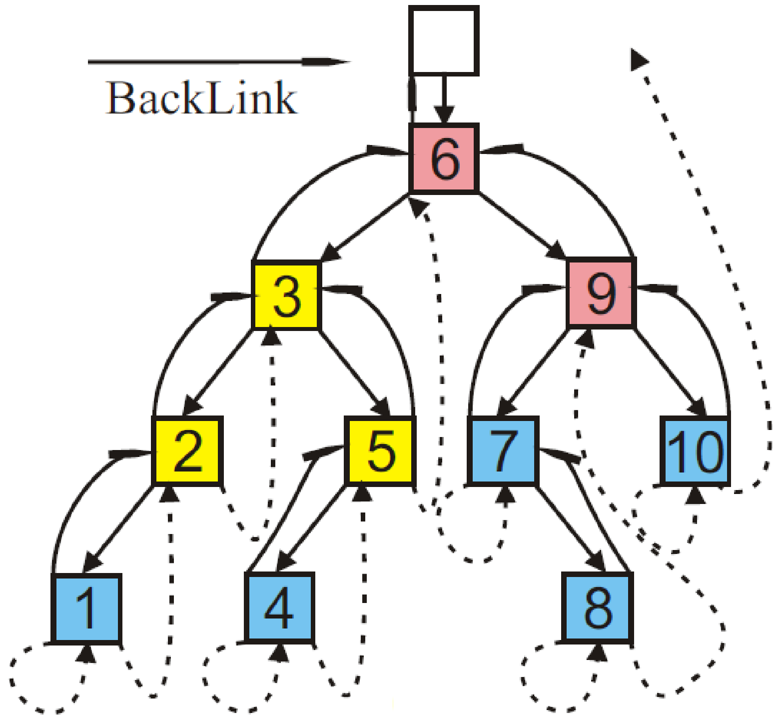
else:

continue;



Add operation

```
bool Add(KType k)
prev = &root[1]; curr = &root[0];
/* Initializing a new node with supplied key and left-link
threaded and pointing to itself.  */
node = new Node(k);
node→child[0] = (node, 0, 0, 1);
while true do
    dir = Locate(prev, curr, k);
    if (dir = 2) then // key exists in the BST
        return false;
    else
        (R, *, *, *) = curr→child[dir];
        /* The located link is threaded.  Set the right-link
        of the adding node to copy this value */
        node→child[1] = (R, 0, 0, 1);
        node→backLink = curr;
        result = CAS(curr→child[dir], (R, 0, 0, 1),
        (node, 0, 0, 0));        // Try inserting the new node.
        if result then  return true;
        else
            /* If the CAS fails, check if the link has been
            marked, flagged or a new node has been inserted.
            If marked or flagged, first help.  */
            (newR, f, m, t) = curr→child[dir];
            if (newR = R) then
                newCurr = prev;
                if m then  CleanMarked(curr, dir);
                else if f then
                    CleanFlagged(curr, R, prev, true);
                curr = newCurr;
                prev = newCurr→backLink;
```



Contributions over State-of-the-art

- Use of backlinks to simplify complexity issue; each thread is a link away from the point of failure.
- Leads to time complexity of $O(h(n) + c)$; first algorithm with additive contention.

7 Steps of Remove operation

- Step 1: Flag the incoming order link.

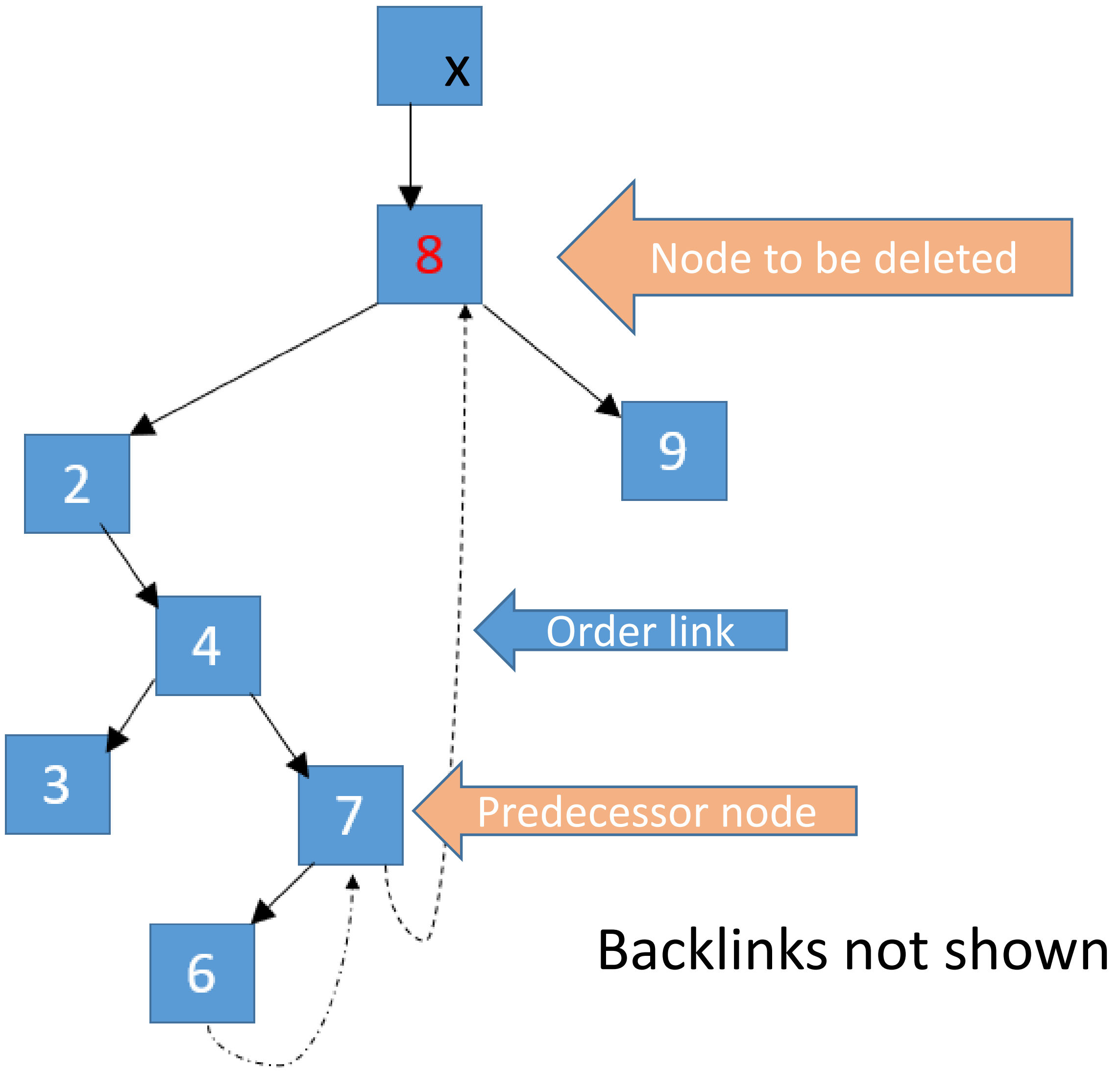
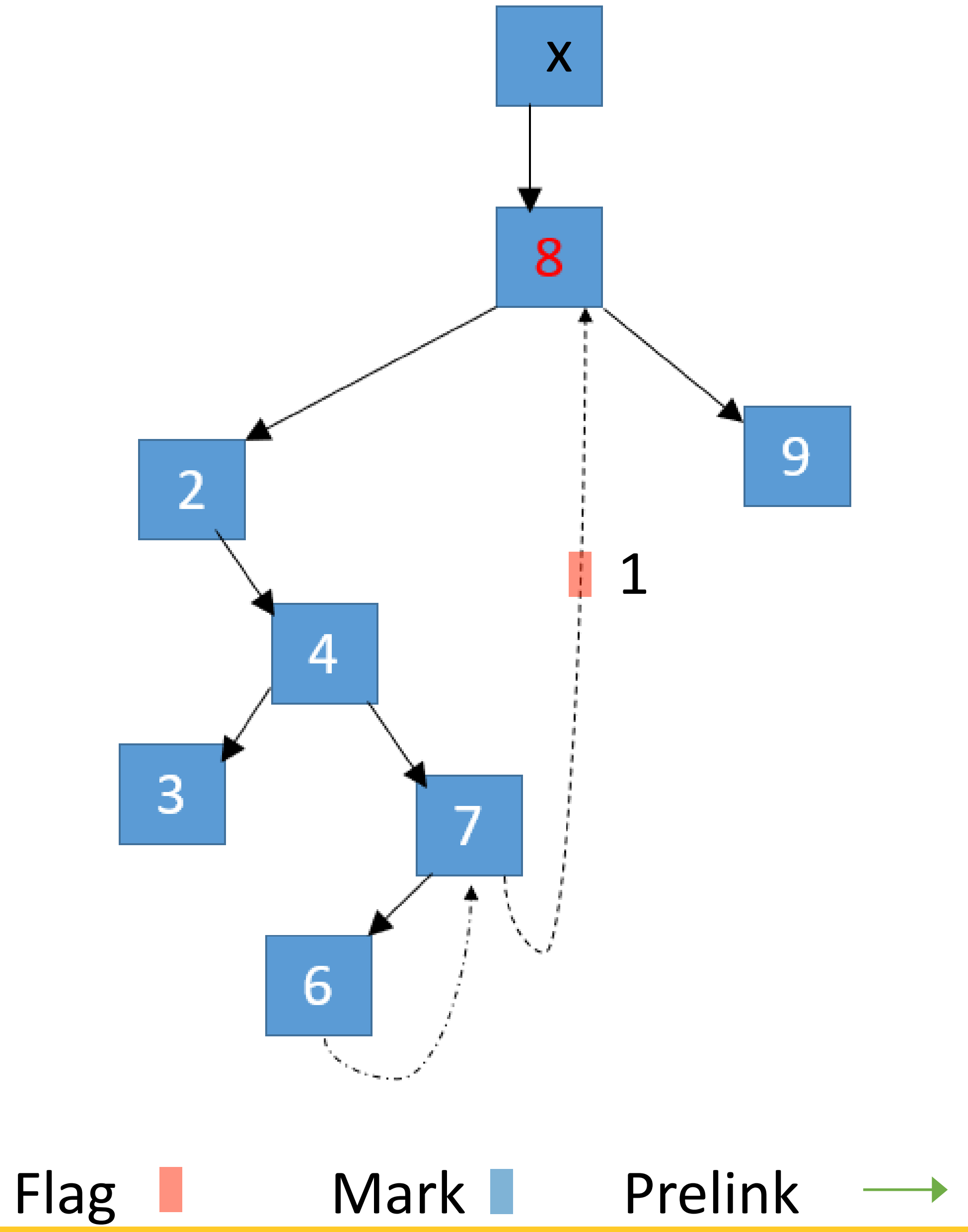


Fig: Removal of Category 3 node (Node 8)

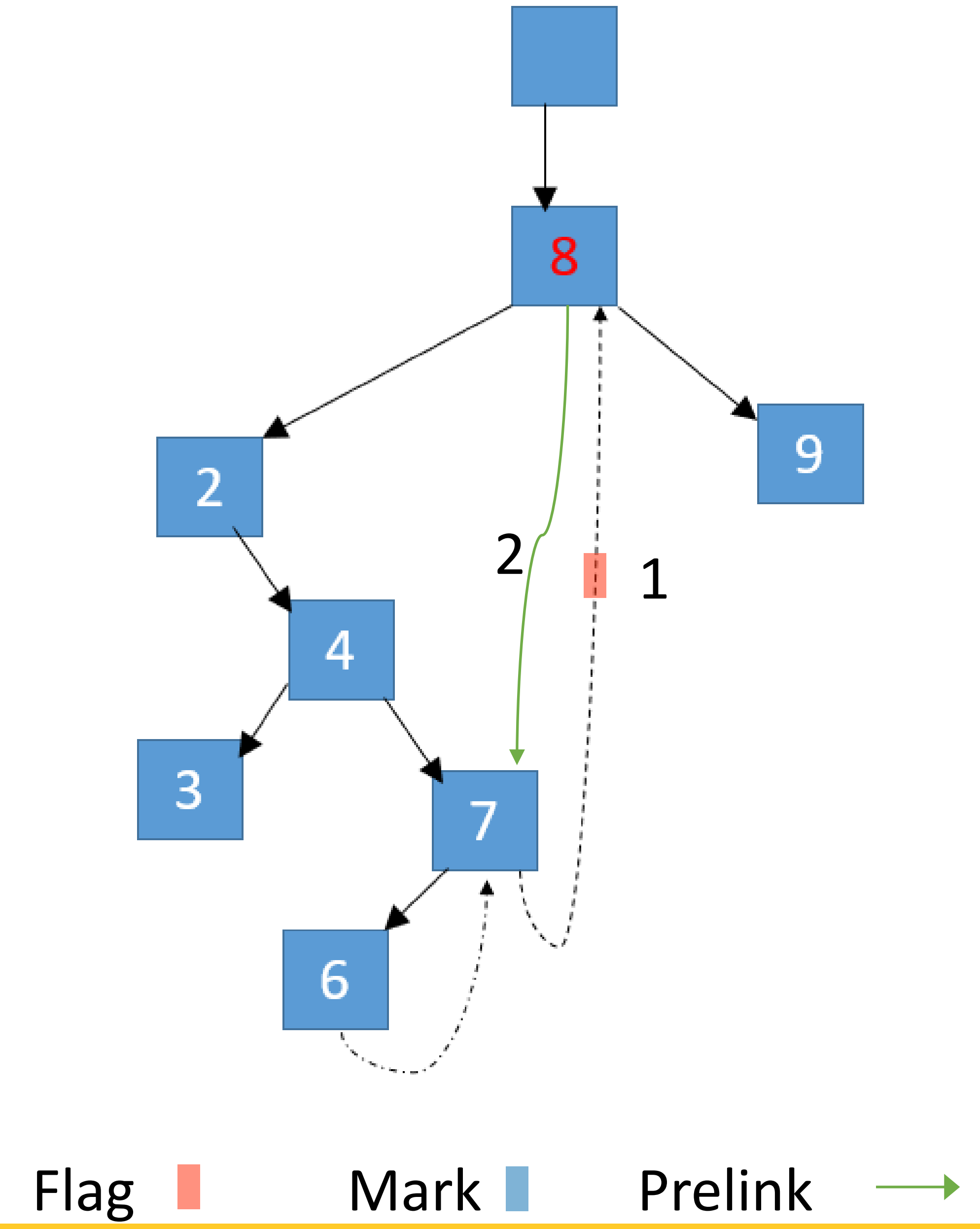
7 Steps of Remove operation

- Step 1: Flag the incoming order link.



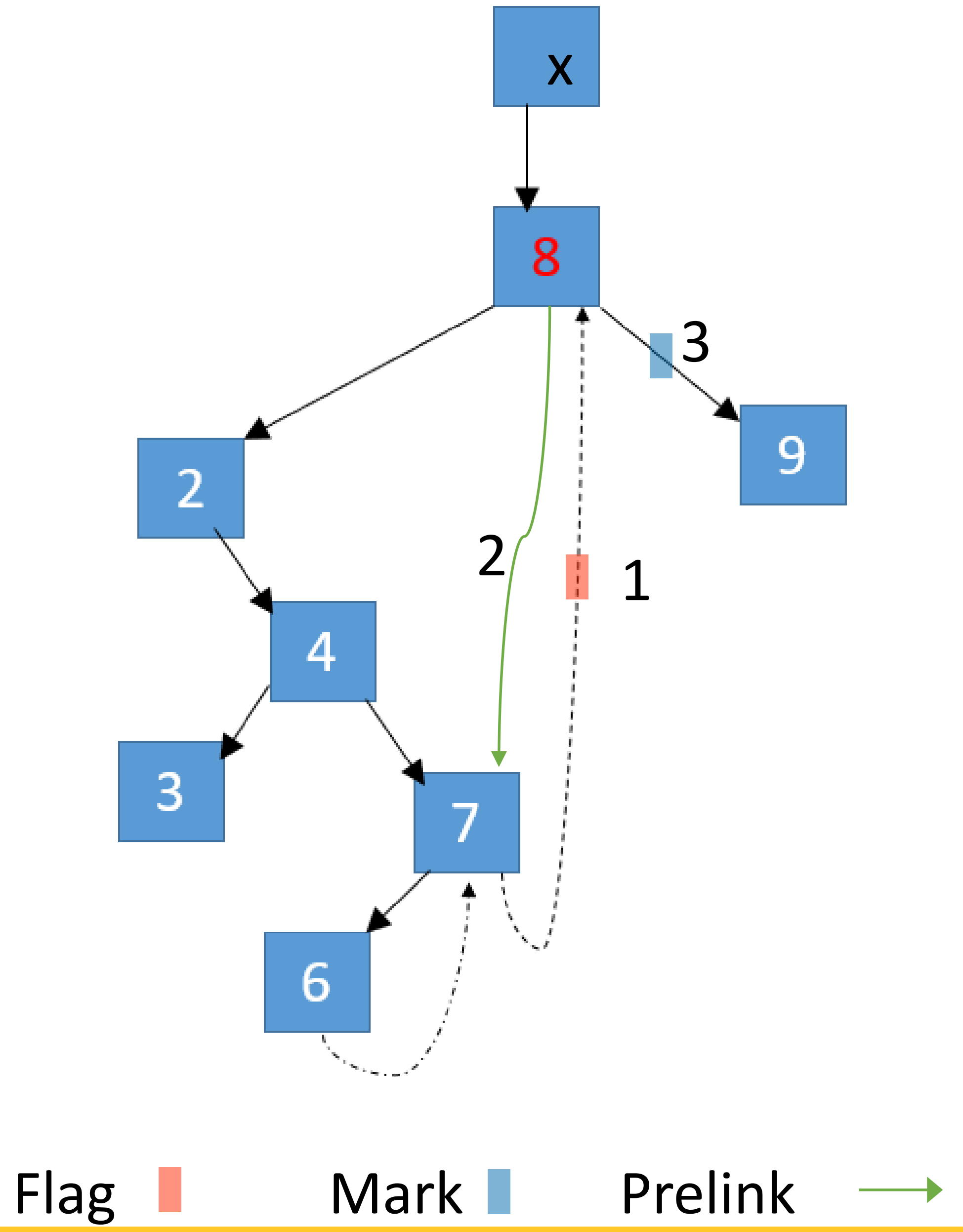
7 Steps of Remove operation

- Step 1: Flag the incoming order link.
- Step 2: Set the prelink



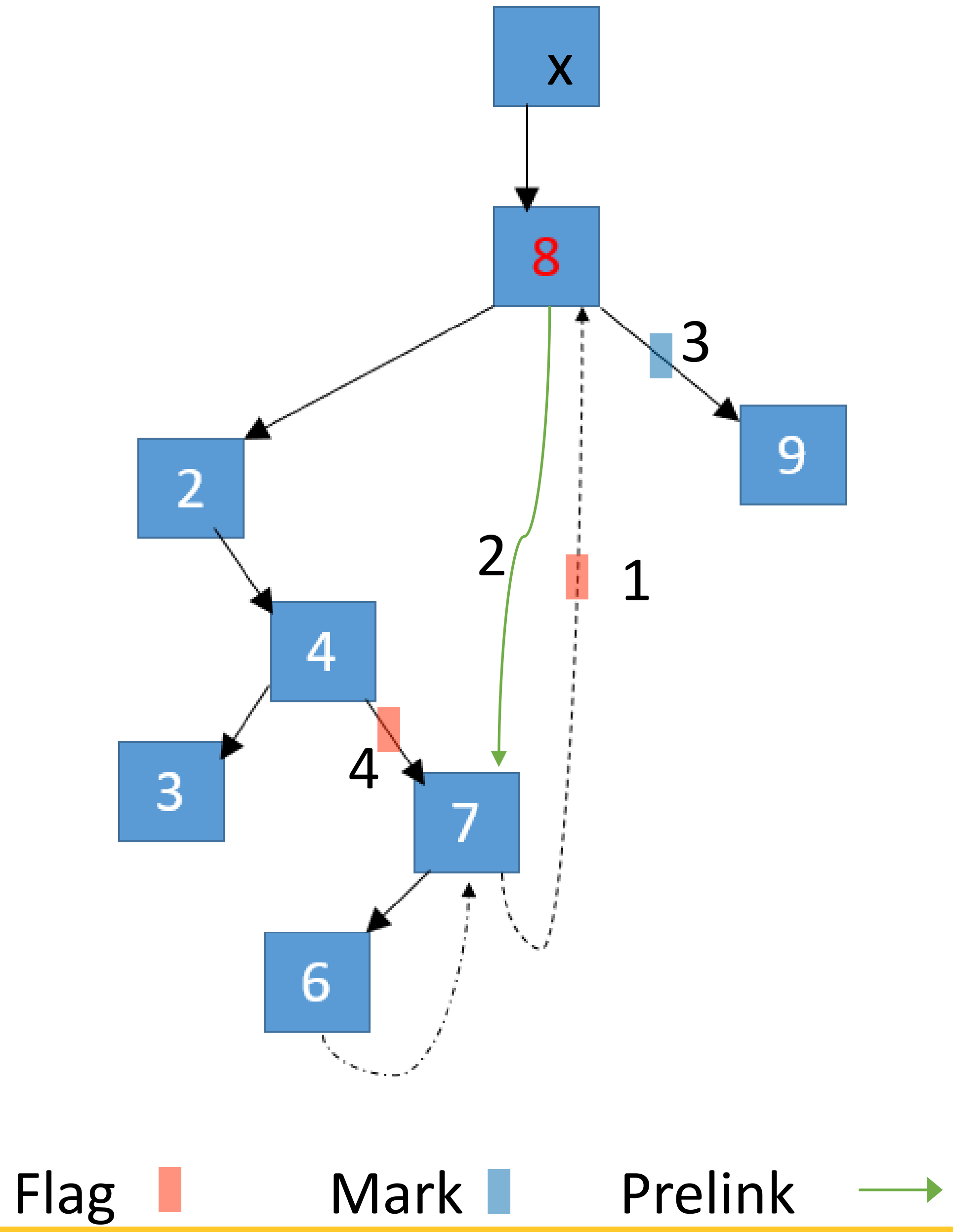
7 Steps of Remove operation

- Step 1: Flag the incoming order link.
- Step 2: Set the prelink
- Step 3: Mark the outgoing right link



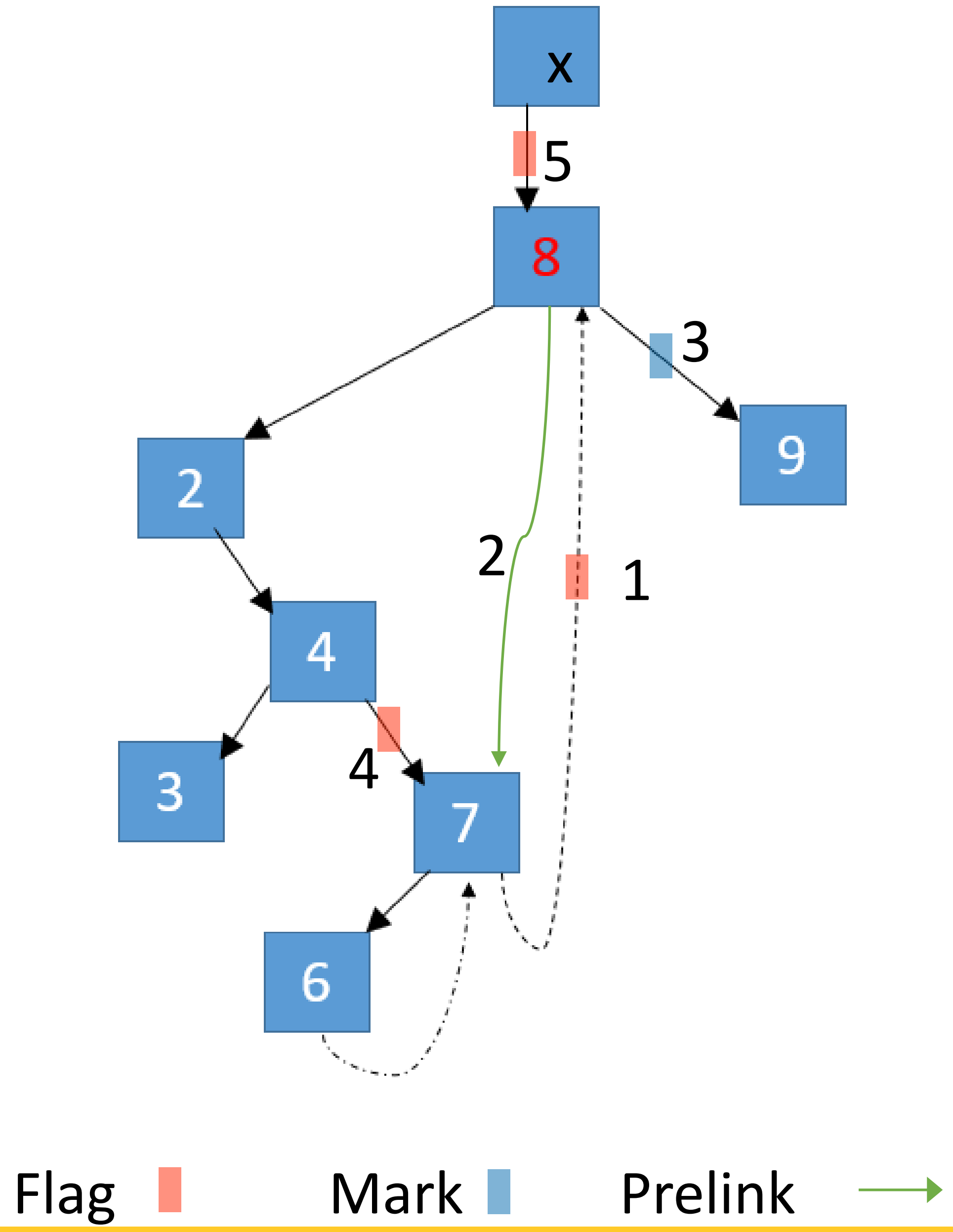
7 Steps of Remove operation

- Step 1: Flag the incoming order link.
- Step 2: Set the prelink
- Step 3: Mark the outgoing right link
- Step 4: Flag the parent-link of the predecessor



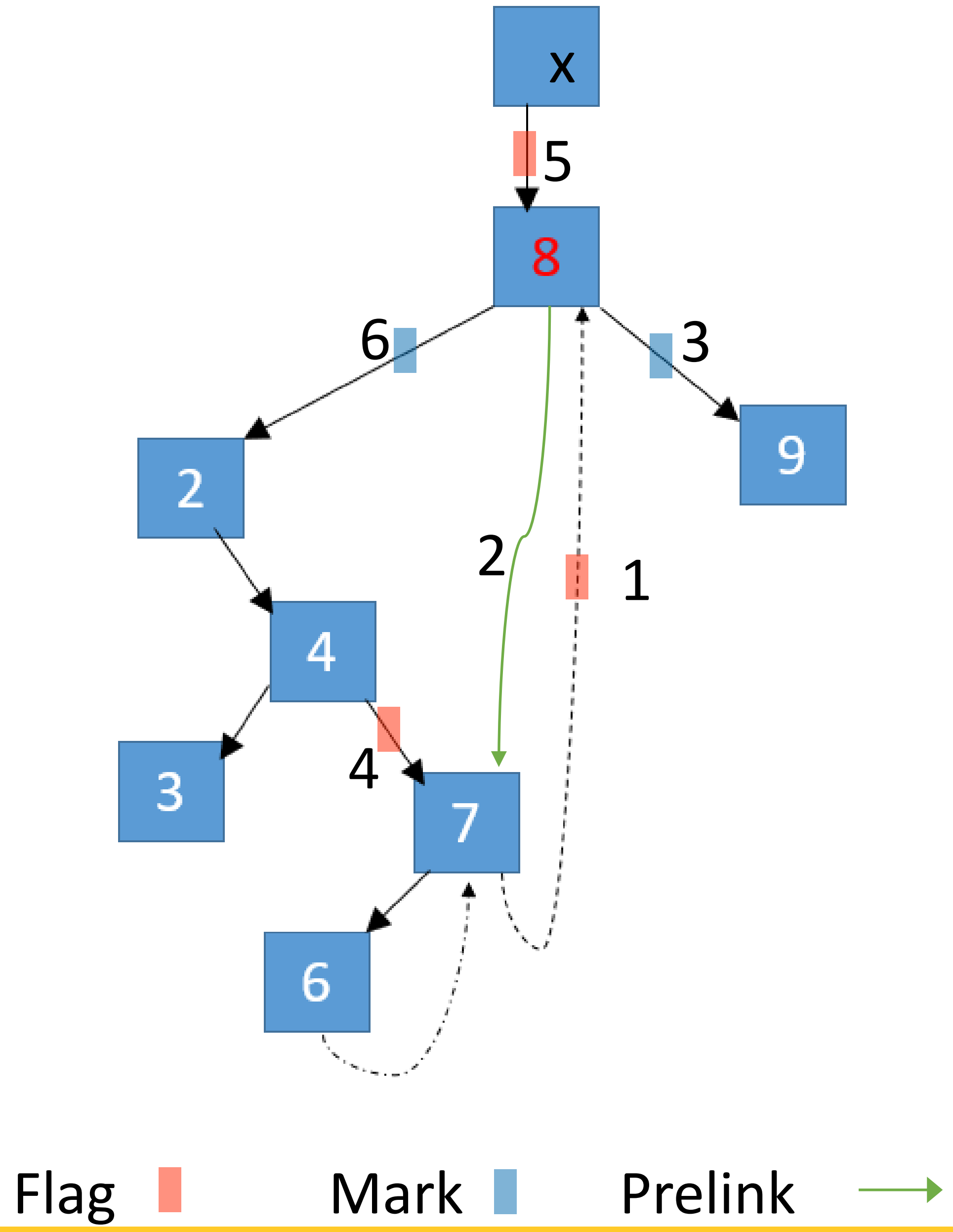
7 Steps of Remove operation

- Step 1: Flag the incoming order link.
- Step 2: Set the prelink
- Step 3: Mark the outgoing right link
- Step 4: Flag the parent-link of the predecessor
- Step 5: Flag the incoming parent link



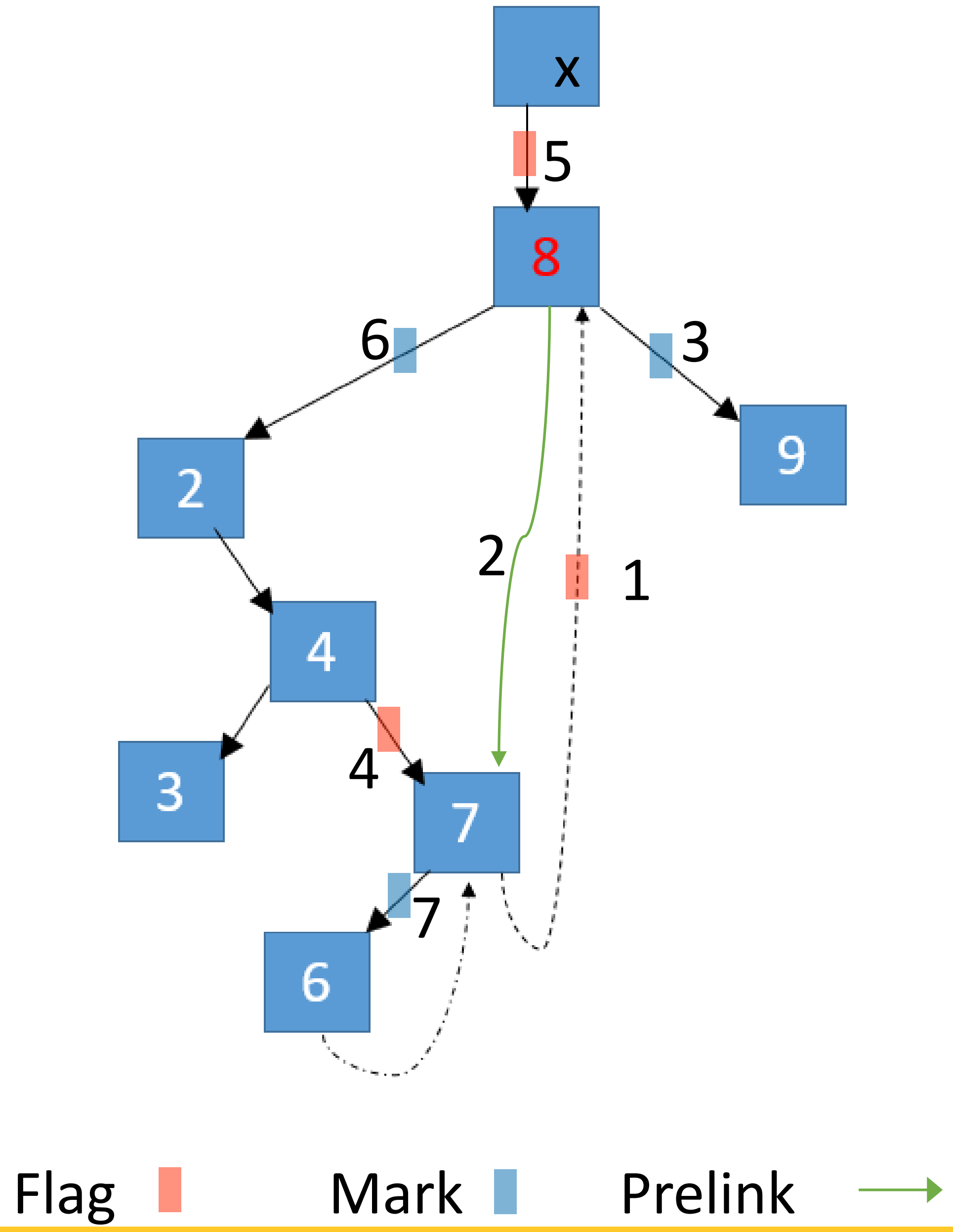
7 Steps of Remove operation

- Step 1: Flag the incoming order link.
- Step 2: Set the prelink
- Step 3: Mark the outgoing right link
- Step 4: Flag the parent-link of the predecessor
- Step 5: Flag the incoming parent link
- Step 6: Mark the outgoing left-link

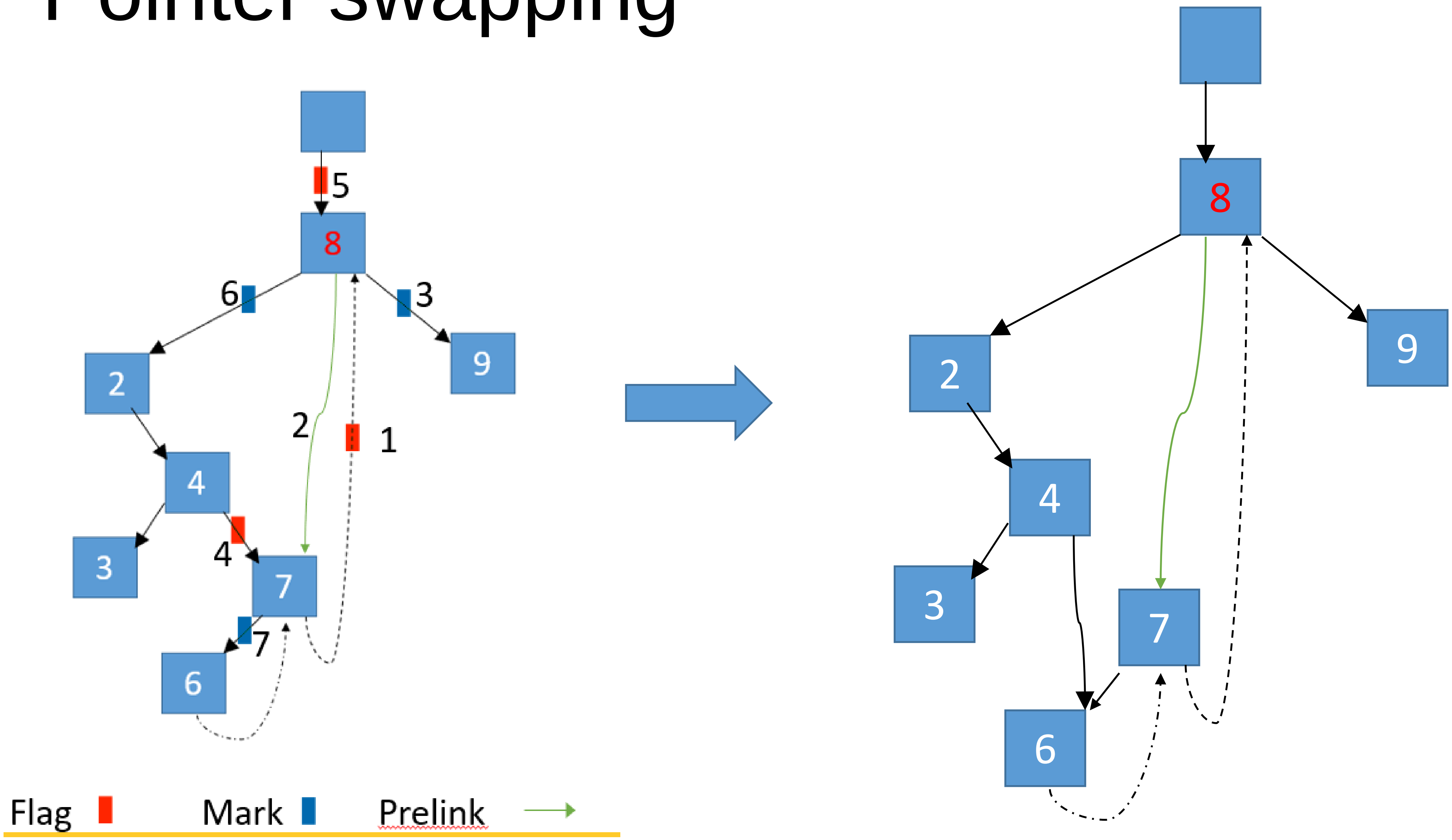


7 Steps of Remove operation

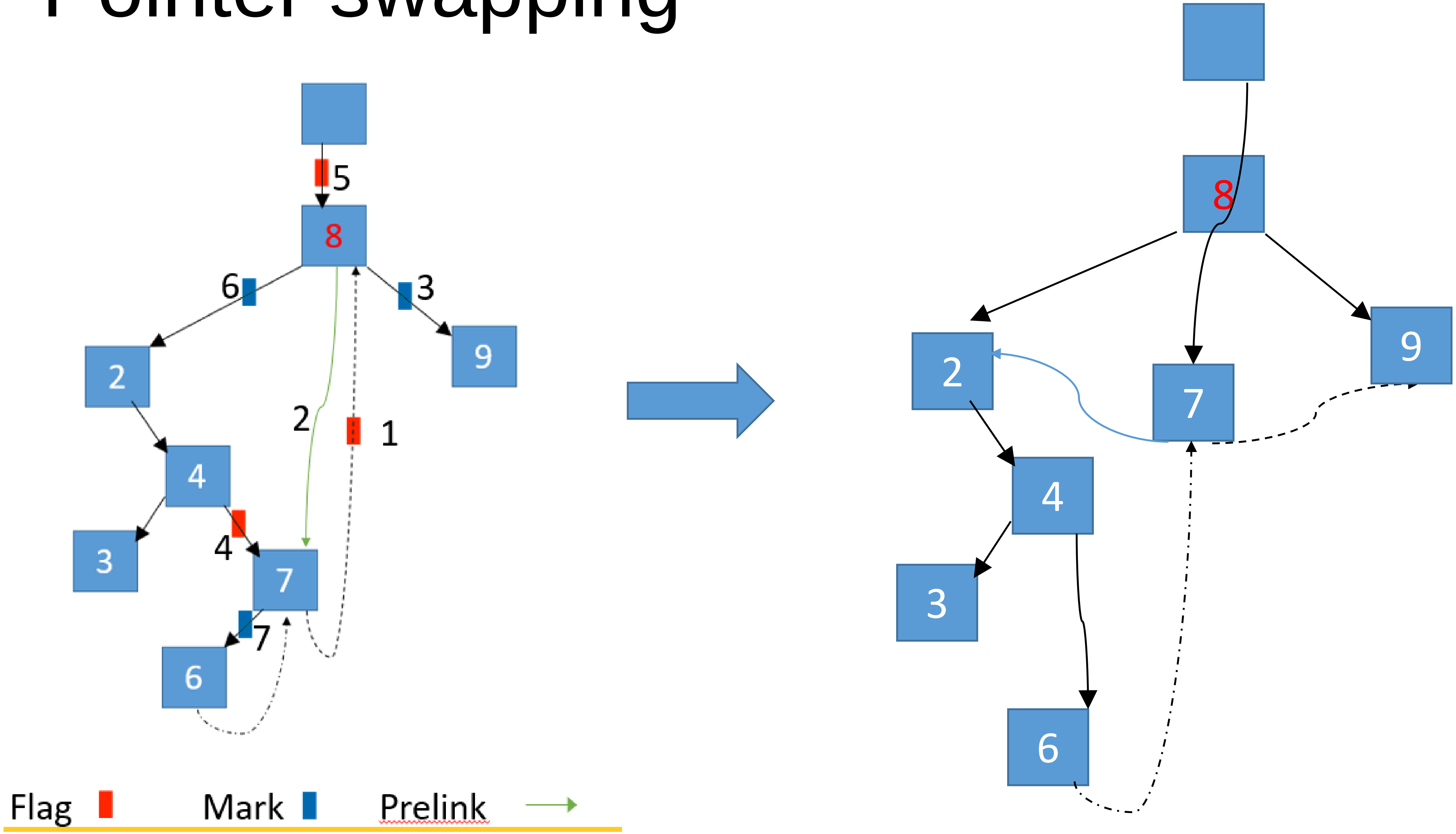
- Step 1: Flag the incoming order link.
- Step 2: Set the prelink
- Step 3: Mark the outgoing right link
- Step 4: Flag the parent-link of the predecessor
- Step 5: Flag the incoming parent link
- Step 6: Mark the outgoing left-link
- Step 7: Mark the outgoing left link of the predecessor



Pointer swapping



Pointer swapping



Why this orderly modification of links?

- This is because, in this way concurrent operation can know in what stage a node is in.
- And the helper function exactly knows what step should be taken to help the operation take step forward.

Remove operation

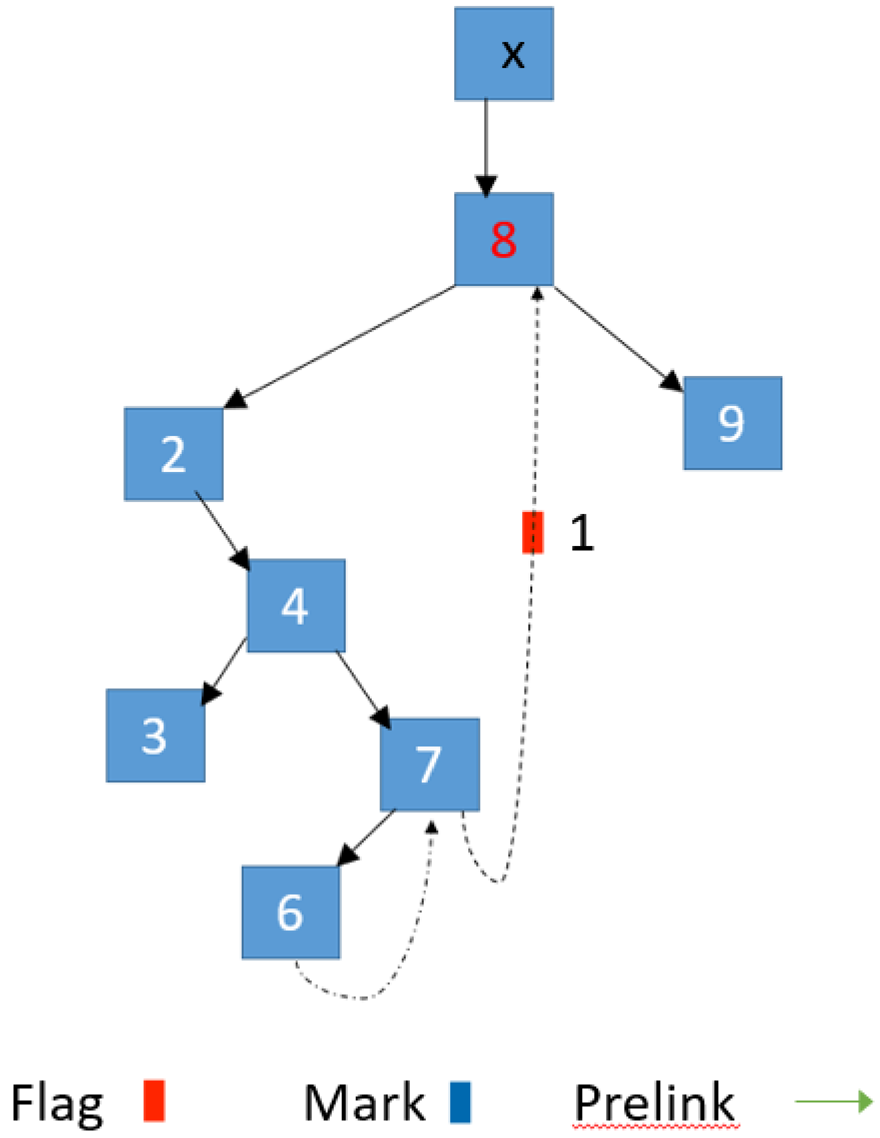
- 4 helper function is used to perform Remove() operation
  - TryFlag()
  - CleanFlag()
  - TryMark()
  - CleanMark()

```
bool REMOVE(KType k)
// Initialize the location variables as before.
prev = &root[1]; curr = &root[0];
dir = LOCATE(prev, curr, k− ε);                // locate
(next, f, ∗, t) = curr→child[dir];
if (k ≠ next→k) then return false;
else
    // flag the order-link
    result = TRYFLAG(curr, next, prev, true);
    if (prev→child[dir].ref = curr) then
        CLEANFLAG(curr, next, prev, true);
return result;
```

CleanFlag()

void CleanFlag(**NPtr&** prev, **NPtr&** curr, **NPtr&** back, **bool** isThread)

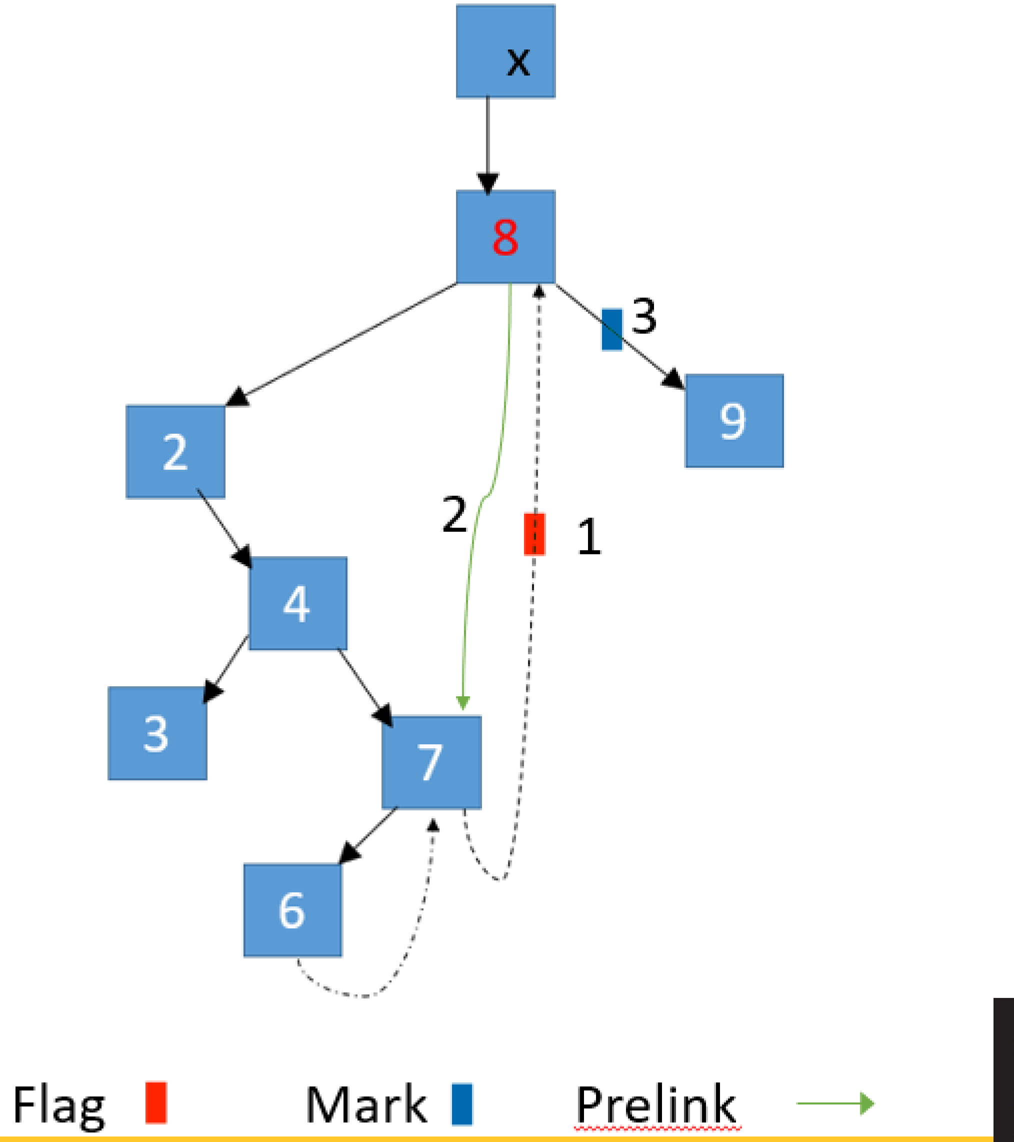
- Helps to perform next step when a link is flagged
- It identifies current state of the concurrent remove operation
- Based on the state of flagged link



CleanFlag()

void CleanFlag(**NPtr&** prev, **NPtr&** curr, **NPtr&** back, **bool** isThread)

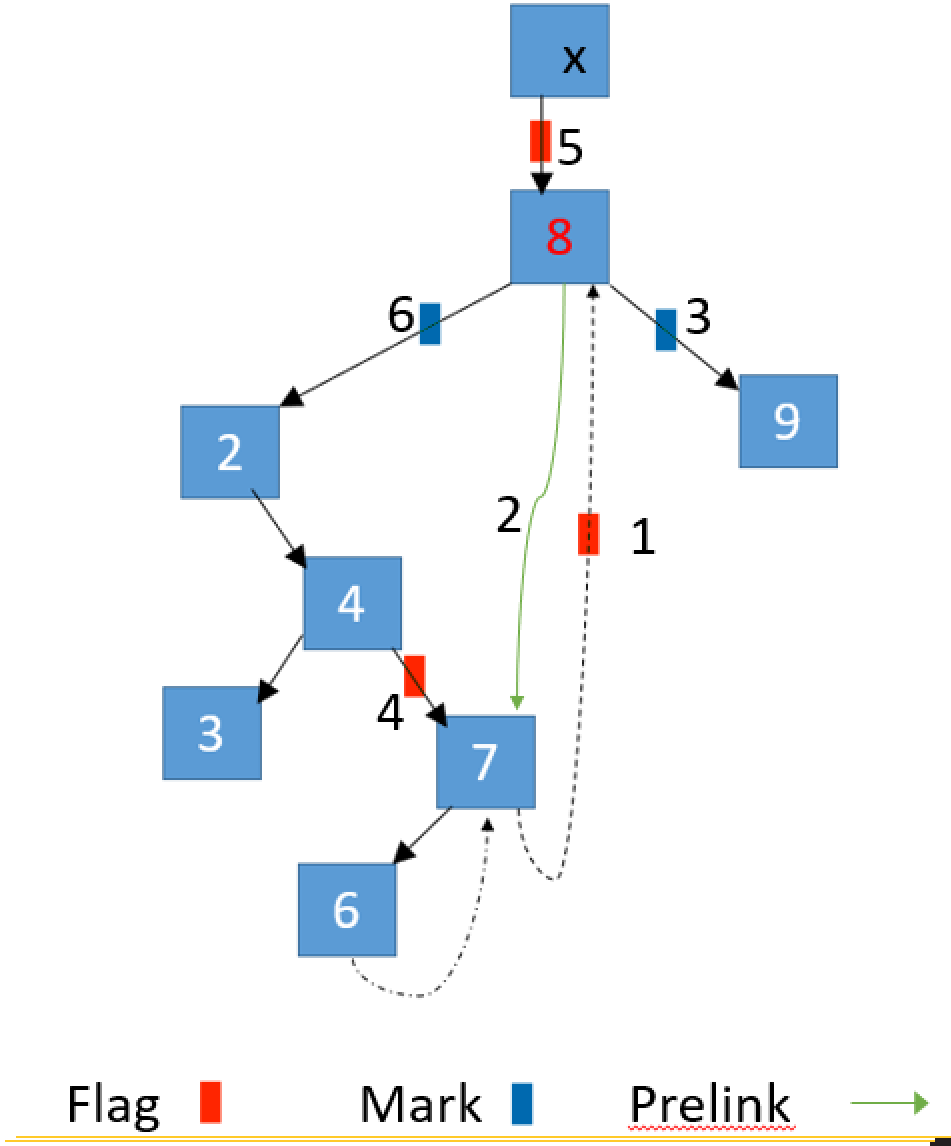
- Link is threaded
  - Now in step 1
  - Perform step 2
    - Set the pre-link
  - Perform step 3
    - Mark the outgoing right-link
  - Then calls CleanMark() to perform next steps



CleanFlag()

void CLEANFLAG(**NPtr&** prev, **NPtr&** curr, **NPtr&** back, **bool** isThread)

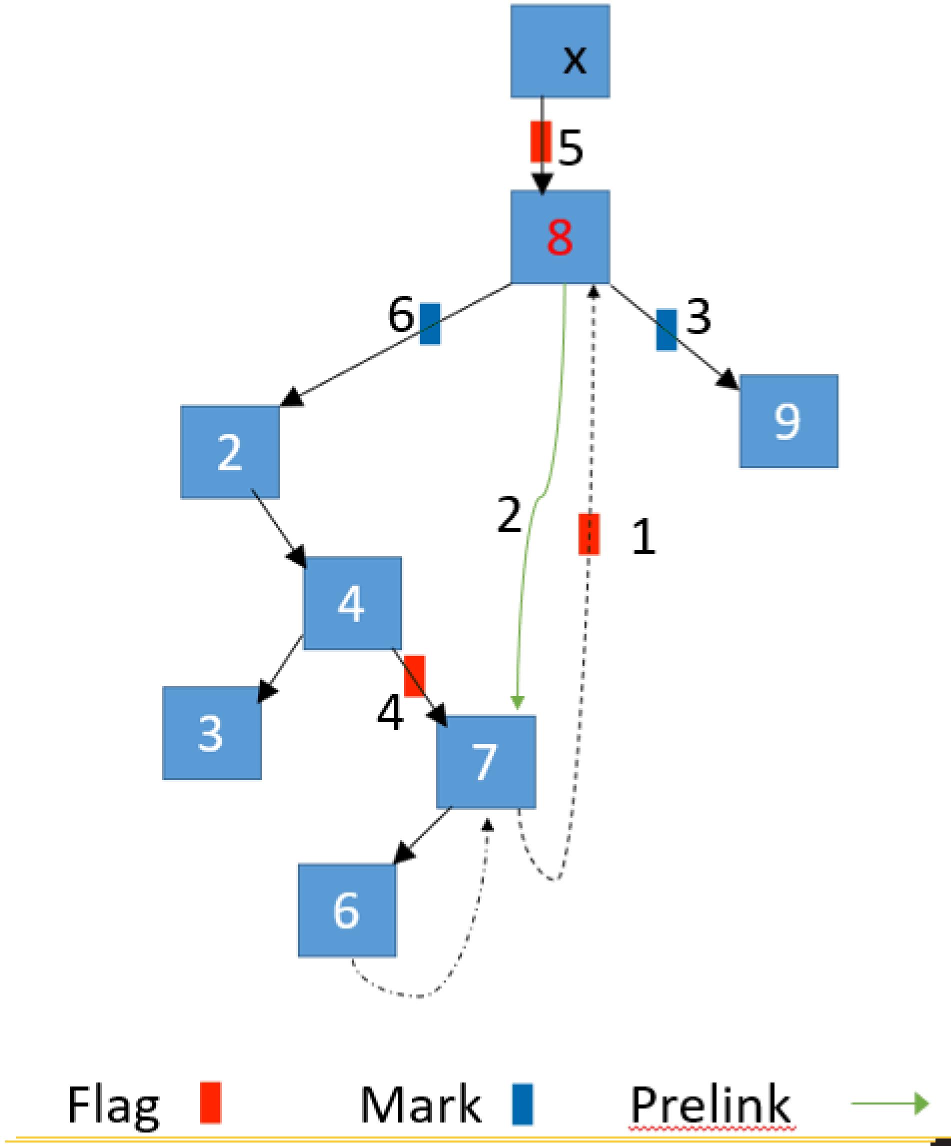
- Link is not threaded
  - It is Parent link
    - Parent link of the predecessor node
      - In step 4
      - Performs step 5: Flag the incoming parent link
    - Parent link of the node to be deleted
      - In step 5
      - Performs step 6: Mark the outgoing left-link
      - CleanMark() is called to perform next steps



CleanFlag()

void CleanFlag(NPtr& prev, NPtr& curr, NPtr& back, bool isThread)

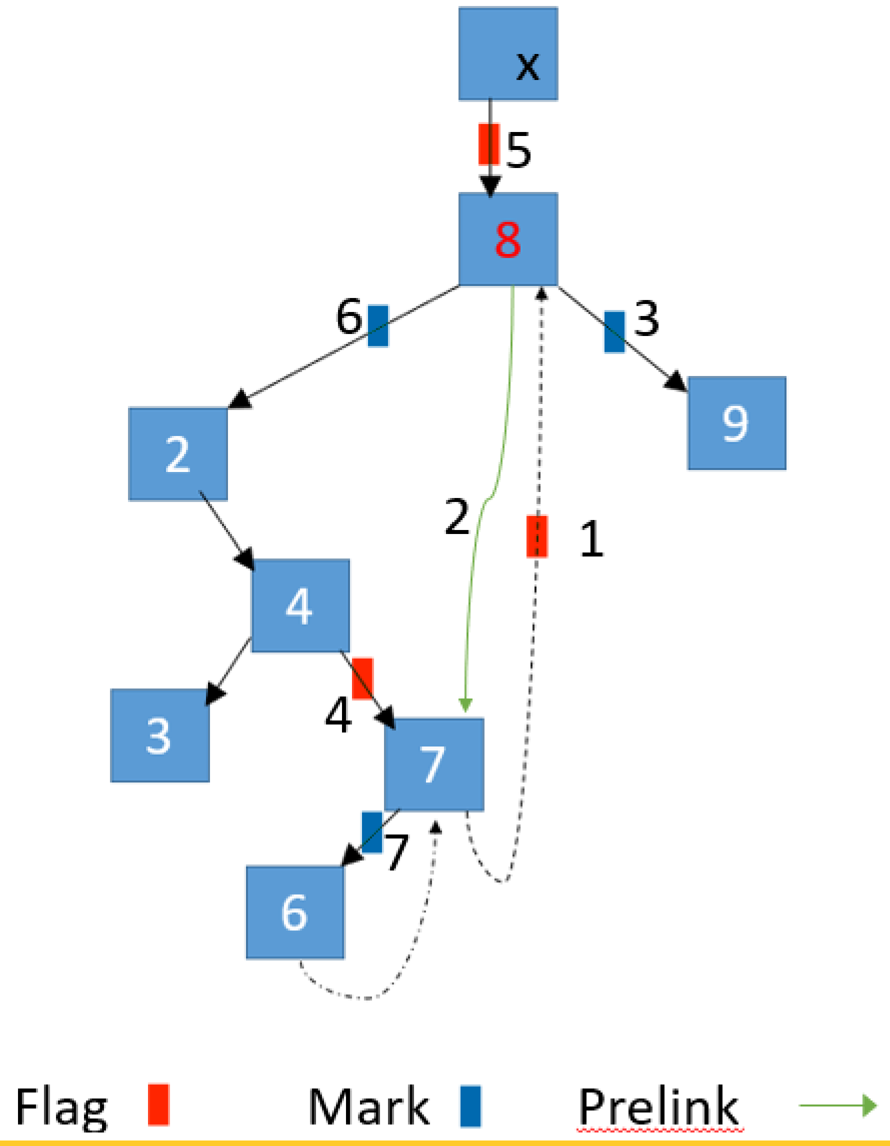
- How CleanFlag() knows which parent link is flagged?
  - If the right child-link of the destination node is
    - threaded and flagged
      - Then, it is parent-link of the predecessor
    - marked
      - Then it is parent-link of the node to be deleted



CleanMark()

void CleanMark(NPtr& curr, int markDir)

- Helps to perform next step when a link is marked
  - If right link is marked
    - In step 3, perform step 4 by calling CleanFlag()
  - If left link is marked
    - In step 6
    - Perform step 7: mark the outgoing left-link of the predecessor
  - If left link is marked and right-link is thread and flagged
    - In step 7
    - Time to perform pointer-swapping.



Our implementation

- For this paper, there is no implementation available
- This paper proposes the idea of bit stealing to store three bits (mark, flag and thread) for each pointer.
  - However, pointer address is architecture dependent.
  - So, bit stealing concept is not portable
- We have created two different node class.
  - Node
  - NodePtrInfo
    - To store child link with 3 bits.

```
class Node{
public:
    Node() noexcept {}
    int k;
    std::atomic<NodePtrInfo> child[2];
    std::atomic<Node*> backLink;
    Node* preLink;
};
```

```
class NodePtrInfo{
public:
    Node* nodeRef;
    bool flag;
    bool mark;
    bool thread;
    NodePtrInfo() noexcept {}
    NodePtrInfo(Node* nodeRef, bool flag, bool mark, bool thread) noexcept  :
                            nodeRef(nodeRef),
                            flag(flag),
                            mark(mark),
                            thread(thread){}
};
```

Correctness

- Logically Removed Node
  - If right link is marked but a parent exists
- Physically Removed Node
  - If no parent exists
- Regular Node
  - All other nodes

Correctness

1. Locate returns true only for non physically removed node.
2. Add operation has to occur at unmarked and unflagged threaded link.
3. An unthreaded link cannot be marked and flagged i.e trymark, tryflag.
4. If a node is logically removed, eventually, it will be physically removed.

Linearizability

- Add Operation
  - Failure: When a key k is already present.
  - Success: When link is not marked, flagged and no node has been inserted.
- Contains Operation
  - Where the comparison between k_curr and k is performed.
- Remove Operation
  - Success: where CAS for swapping flagged parent link(step V) is performed.
  - Failure:
    - If node is not located -> LP of unsuccessful LOCATE.
    - If node is located -> LP of concurrent remove that flags the order link.

Challenges

- Experimental evaluation not available
  - Next best thing?
- Major bugs in the pseudocode.
  - e.g incorrect function calls, no declarations, incorrect variables used, typos, incorrect function parameters.

Conclusion

- Exploit multiple link pointers for a node.
  - Challenging and complex to implement. Other approaches use simple ideas to achieve lock freedom.
    - Use of search path and store locally.
- Remove operation quite expensive; up to 4 pointers need to be modified in the worst case.

References

- Lock Free Binary Search Trees

https://arxiv.org/abs/1404.3272

- BST Image

https://upload.wikimedia.org/wikipedia/commons/d/da/Binary_search_tree.svg

- BST operations

https://www.techiedelight.com/deletion-from-bst/

https://www.techiedelight.com/search-given-key-in-bst/

https://www.techiedelight.com/insertion-in-bst/