Concurrent Data Structures and Robust Shared Memory Building Blocks

Philipp Woelfel & Panagiota (Youla) Fatourou
University of Crete, Department of Computer Science
Foundation for Research and Technology - Hellas

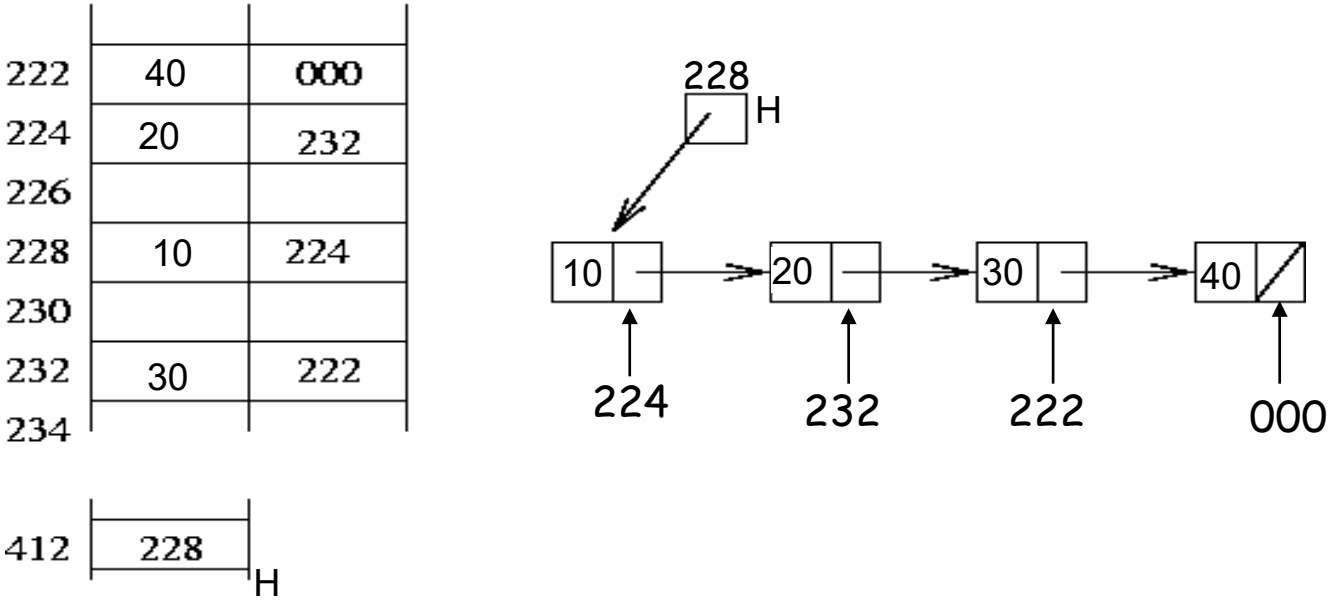
Distributed Computing Mexico Summer School
June 2025, Huatulco, Mexico

Set

- A set stores unique keys and supports the following operations:
  - **insert**(int key): inserts key to the set (if key not already in set); returns FALSE if key already in the set and TRUE otherwise
  - **delete**(int key): deletes key from the set (if key is in the set); returns FALSE if key is not in the set and TRUE otherwise
  - **search**(int key): searches for key and returns TRUE if it is in the set and FALSE otherwise.
- An operation is called successful if it returns TRUE and unsuccessful if it returns FALSE.

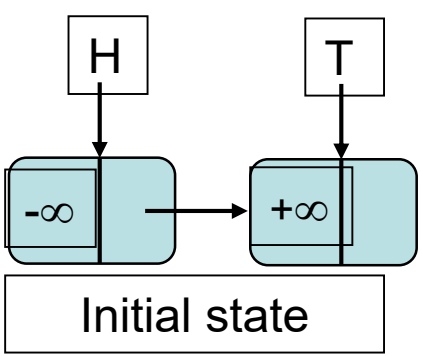
Dynamic List to Implement a Set

Lewis & Denenberg, Data Structures & Their Algorithms

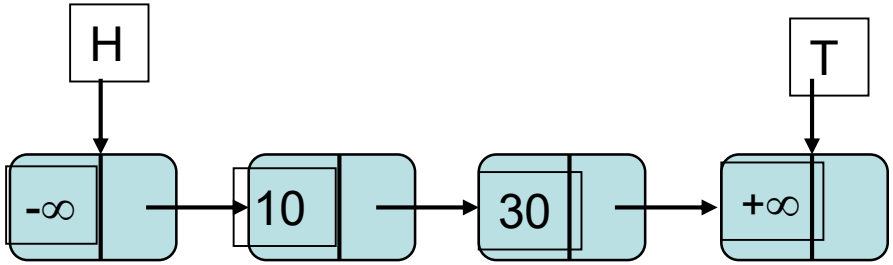


- Each node has two fields:
  - key: integer
  - next: pointer to next node of the list.

Sorted Linked List with Sentinel Nodes



Initial state



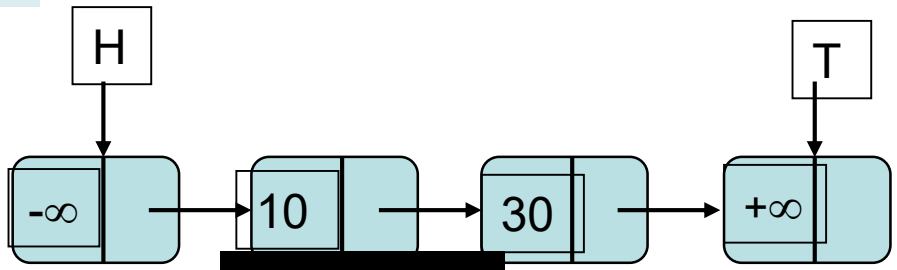
Nodes with keys 10 and 30 are regular

- The list contains regular nodes and two sentinel nodes, called **Head** (H) and **Tail** (T), that point to the first and last element of the list, respectively.
- Sentinel nodes are initially in the list and are never deleted; the key of node pointed to by H is MININT (- ∞) and the key of node pointed to by T is MAXINT (+ ∞).
- H and T never change.

Sequential Search

```
boolean search(int key) {
     NODE *curr; boolean result;

    2. curr = H;
    3. while (curr->key < key)
    4.       curr = curr->next;
    5. if (key == curr->key)
    6.         result = TRUE;
    7. else result = FALSE;

    9. return result;
}
```

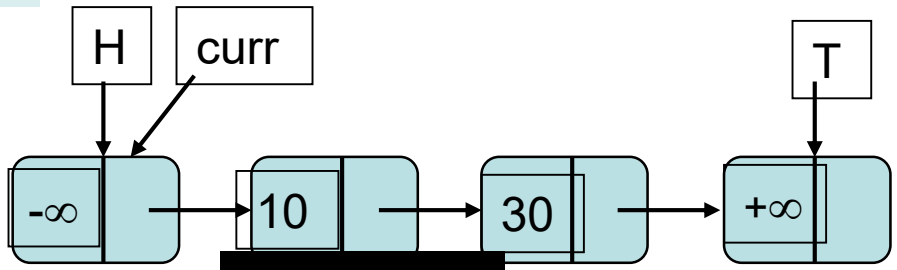
Search(20)



Sequential Search

```
boolean search(int key) {
     NODE *curr; boolean result;

    2. curr = H;
    3. while (curr->key < key)
    4.       curr = curr->next;
    5. if (key == curr->key)
    6.         result = TRUE;
    7. else result = FALSE;

    9. return result;
}
```

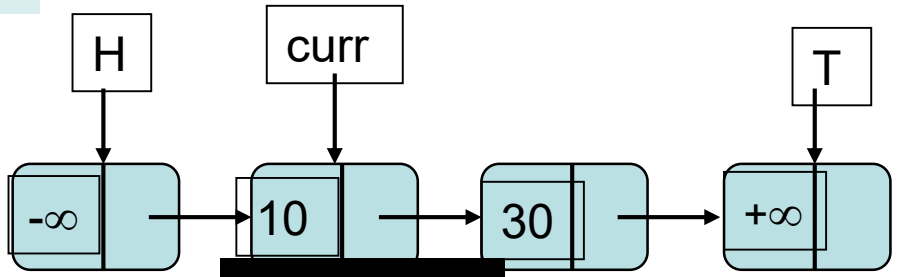
Search(20)



Sequential Search

```
boolean search(int key) {
     NODE *curr; boolean result;

    2. curr = H;
    3. while (curr->key < key)
    4.       curr = curr->next;
    5. if (key == curr->key)
    6.         result = TRUE;
    7. else result = FALSE;

    9. return result;
}
```

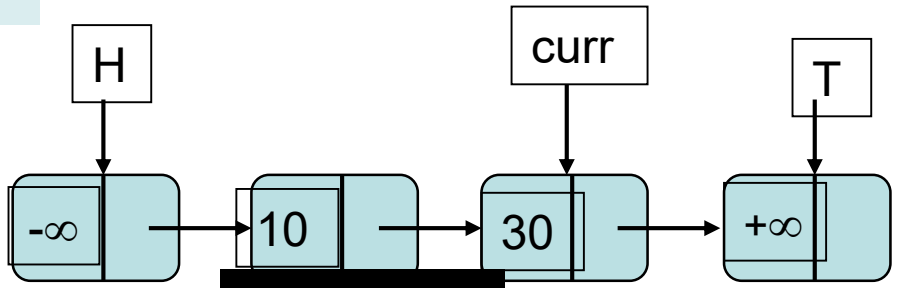
Search(20)



Sequential Search

```
boolean search(int key) {
     NODE *curr; boolean result;

    2. curr = H;
    3. while (curr->key < key)
    4.       curr = curr->next;
    5. if (key == curr->key)
    6.         result = TRUE;
    7. else result = FALSE;

    9. return result;
}
```

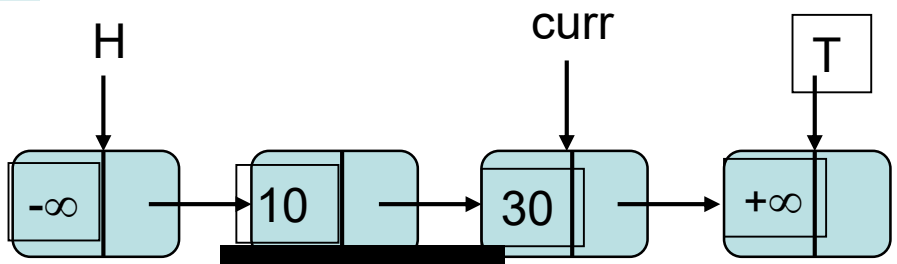
Search(20)



Sequential Search

```
boolean search(int key) {
     NODE *curr; boolean result;

    2. curr = H;
    3. while (curr->key < key)
    4.       curr = curr->next;
    5. if (key == curr->key)
    6.         result = TRUE;
    7. else result = FALSE;

    9. return result;
}
```

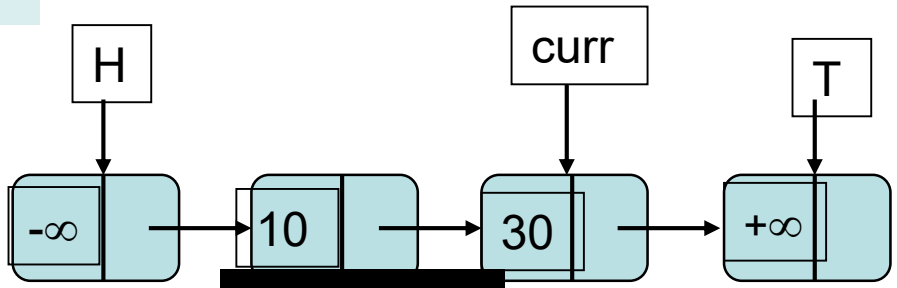
Search(20)



Sequential Search

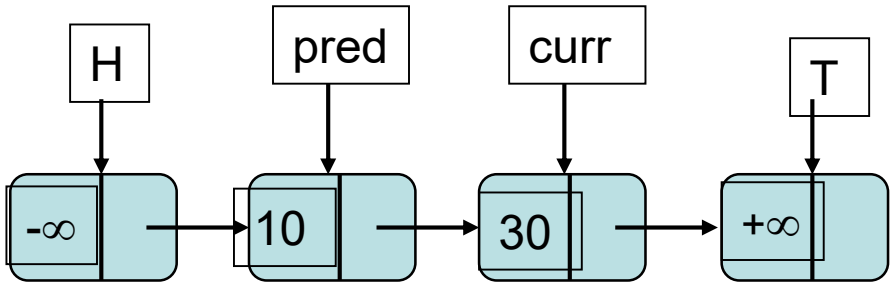
```
boolean search(int key) {
     NODE *curr; boolean result;

    2. curr = H;
    3. while (curr->key < key)
    4.       curr = curr->next;
    5. if (key == curr->key)
    6.         result = TRUE;
    7. else result = FALSE;

    9. return result;
}
```

Search(20) → FALSE



Sequential Insert

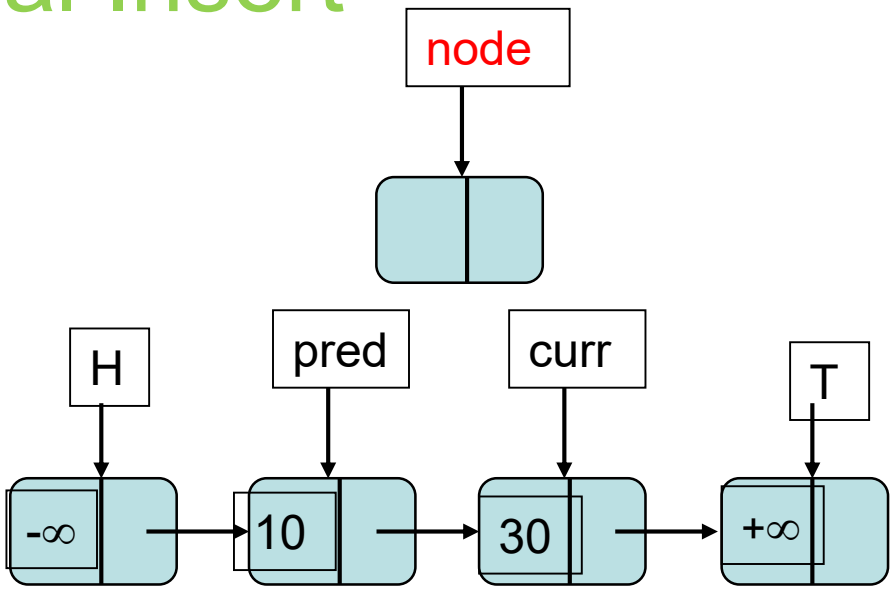
```
boolean insert(int key, T x) {
// code for process p
    Node *pred, *curr;
    boolean result;

    11. pred = H;
    12. curr = pred->next;
    13. while (curr->key < key) {
    14.       pred = curr;
    15.       curr = curr->next;
    }
    16. if (key == curr->key)
          result = FALSE;
    17. else {
    18.     NODE *node = newcell(NODE);
    19.     node->next = curr;
    20.    node->key = key;
    21.     pred->next = node;
    22.    result = TRUE;
    }

    24. return result;
}
```



Insert(20)

Sequential Insert

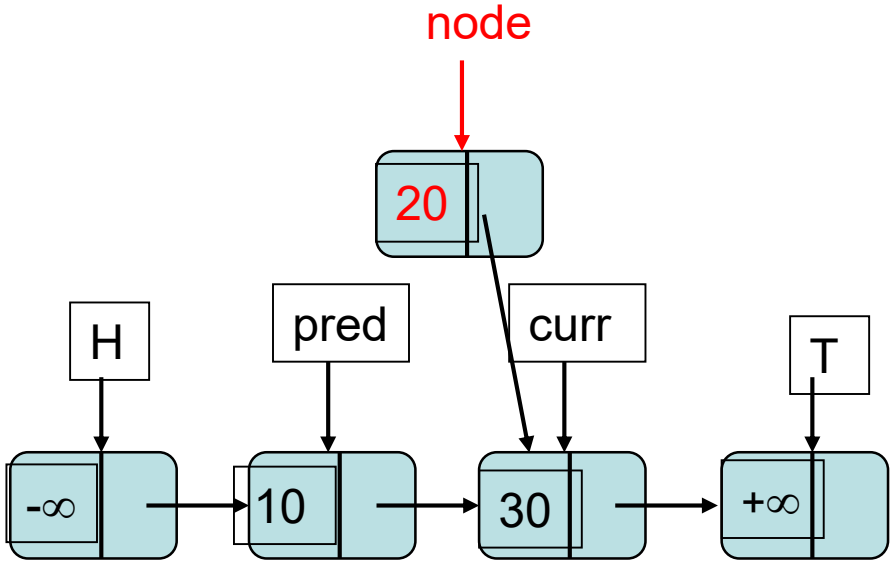
```
boolean insert(int key, T x) {
// code for process p
    Node *pred, *curr;
    boolean result;

    11. pred = H;
    12. curr = pred->next;
    13. while (curr->key < key) {
    14.     pred = curr;
    15.     curr = curr->next;
    }
    16. if (key == curr->key)
          result = FALSE;
    17. else {
    18.    NODE *node = newcell(NODE);
    19.    node->next = curr;
    20.    node->key = key;
    21.    pred->next = node;
    22.    result = TRUE;
    }

    24. return result;
}
```



Insert(20)

Sequential Insert

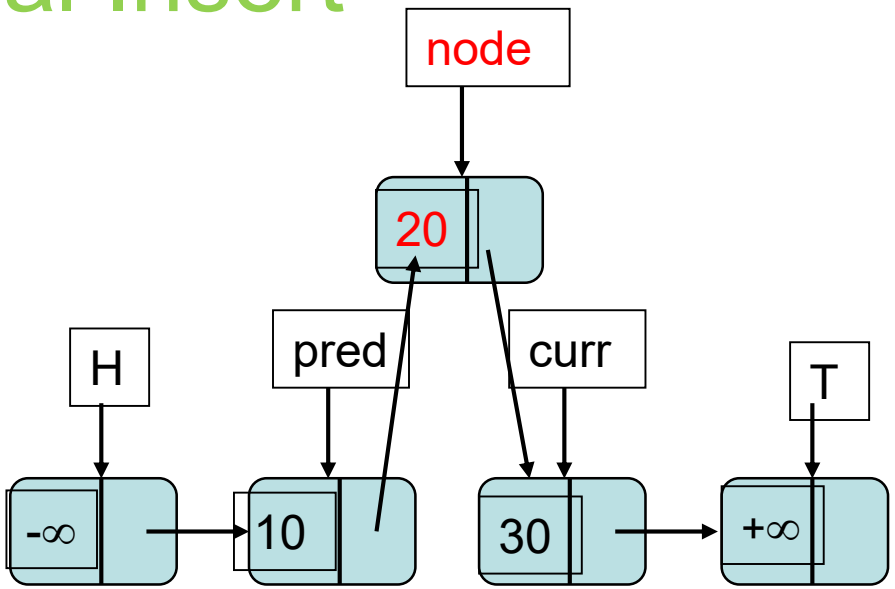
```
boolean insert(int key, T x) {
// code for process p
    Node *pred, *curr;
    boolean result;

    11. pred = H;
    12. curr = pred->next;
    13. while (curr->key < key) {
    14.     pred = curr;
    15.     curr = curr->next;
    }
    16. if (key == curr->key)
          result = FALSE;
    17. else {
    18.    NODE *node = newcell(NODE);
    19.    node->next = curr;
    20.    node->key = key;
    21.    pred->next = node;
    22.    result = TRUE;
    }

    24. return result;
}
```



Insert(20)

Sequential Insert

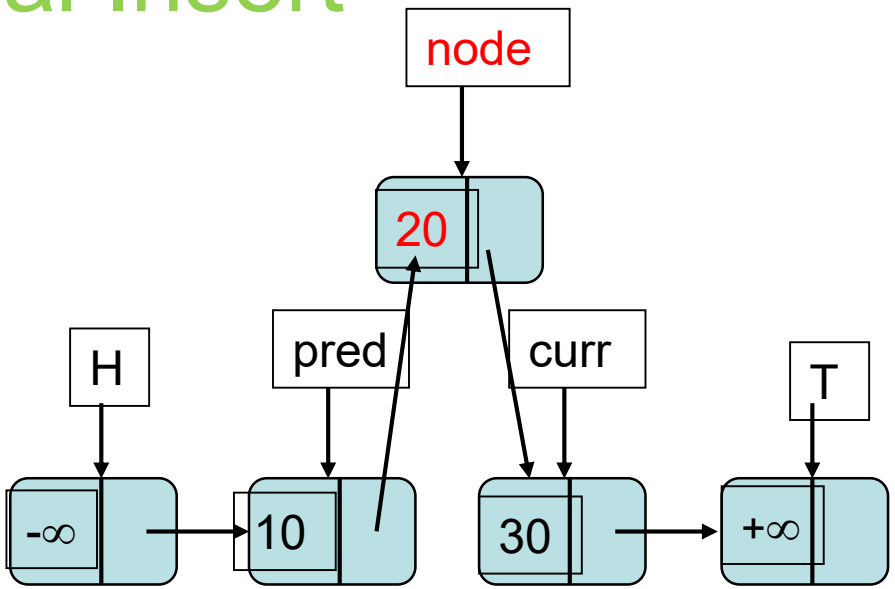
```
boolean insert(int key, T x) {
// code for process p
    Node *pred, *curr;
    boolean result;

    11. pred = H;
    12. curr = pred->next;
    13. while (curr->key < key) {
    14.     pred = curr;
    15.     curr = curr->next;
    }
    16. if (key == curr->key)
          result = FALSE;
    17. else {
    18.    NODE *node = newcell(NODE);
    19.    node->next = curr;
    20.   node->key = key;
    21.    pred->next = node;
    22.   result = TRUE;
    }

    24. return result;
}
```



Insert(20)

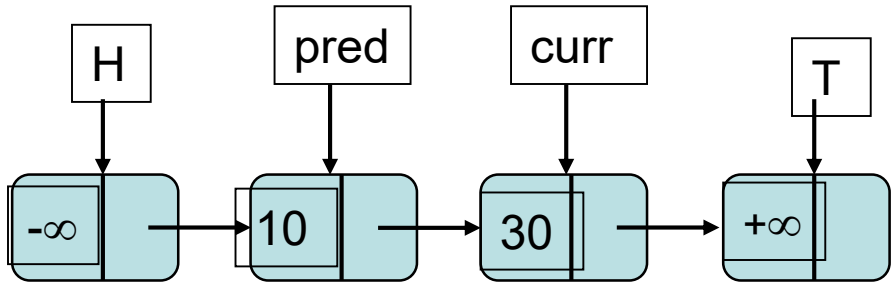
Sequential Insert

```
boolean insert(int key, T x) {
// code for process p
    Node *pred, *curr;
    boolean result;

    11. pred = H;
    12. curr = pred->next;
    13. while (curr->key < key) {
    14.     pred = curr;
    15.     curr = curr->next;
    }
    16. if (key == curr->key)
          result = FALSE;
    17. else {
    18.     NODE *node = newcell(NODE);
    19.     node->next = curr;
    20.    node->key = key;
    21.     pred->next = node;
    22.    result = TRUE;
    }

    24. return result;
}
```



Insert(20) → TRUE

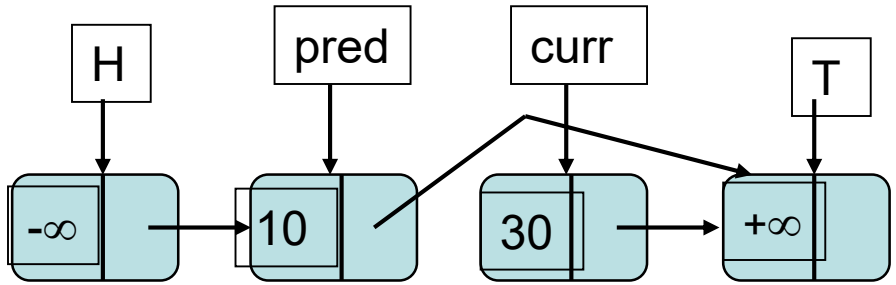
Sequential Delete



Delete(30)

```
boolean delete(int key) {
    // code for process p
    Node *pred, *curr;
    boolean result;

    26. pred = H;
    27. curr = pred->next;
    28. while (curr->key < key) {
    29.     pred = curr;
    30.     curr = curr->next;
        }
    31. if (key == curr->key) {
    32.     pred->next = curr->next;
    33.     result = TRUE;
        }
    34. else result = FALSE;

    36.  return result;
}
```

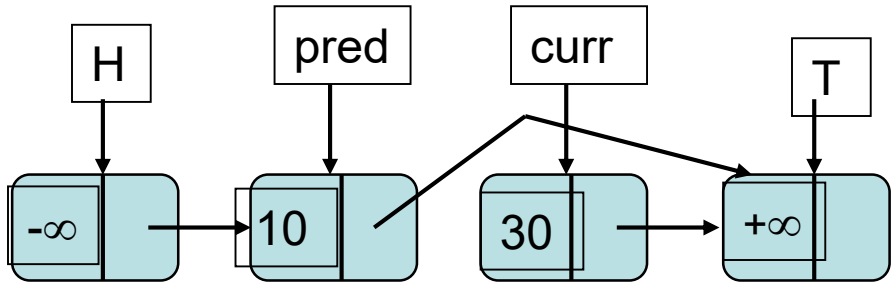
Sequential Delete



Delete(30)

```
boolean delete(int key) {
    // code for process p
    Node *pred, *curr;
    boolean result;

    26. pred = H;
    27. curr = pred->next;
    28. while (curr->key < key) {
    29.     pred = curr;
    30.     curr = curr->next;
        }
    31. if (key == curr->key) {
    32.     pred->next = curr->next;
    33.     result = TRUE;
        }
    34. else result = FALSE;

    36.  return result;
}
```

Sequential Delete



Delete(30) → TRUE

```
boolean delete(int key) {
    // code for process p
    Node *pred, *curr;
    boolean result;

    26. pred = H;
    27. curr = pred->next;
    28. while (curr->key < key) {
    29.     pred = curr;
    30.     curr = curr->next;
        }
    31. if (key == curr->key) {
    32.     pred->next = curr->next;
    33.     result = TRUE;
        }
    34. else result = FALSE;

    36.  return result;
}
```

Homework

1. Draw a linked list with >6 regular nodes.
2. Trace the execution of a successful and an unsuccessful Search operation.
3. Trace the execution of two successful Insert operations.
4. Trace the execution of two successful Delete operations.
5. Trace the execution of an unsuccessful Insert and an unsuccessful Delete operation.
6. Draw (abstract and more detailed) figures to illustrate the traces.

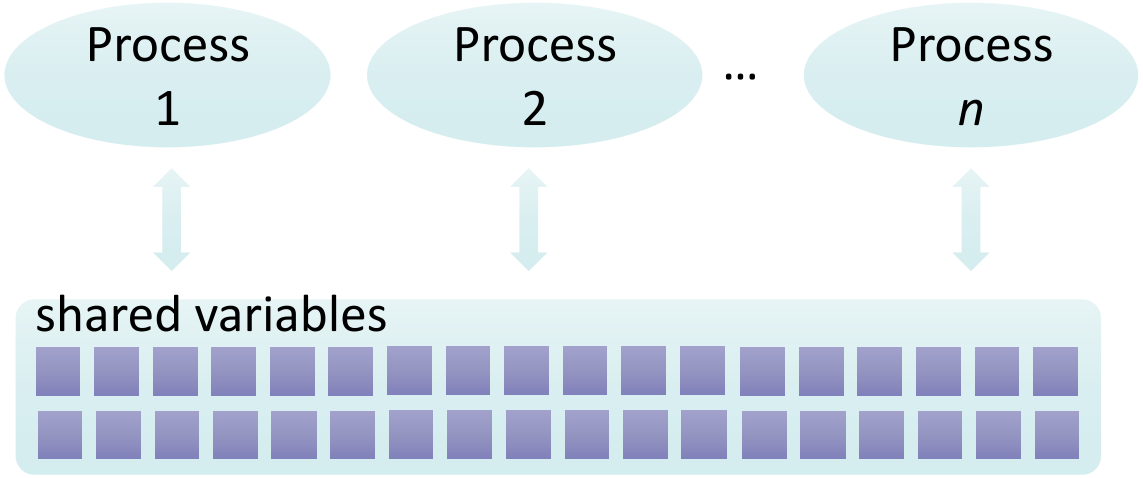
HAR.S.H. HARdware-aware Similarity search
Greece 2.0 NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union NextGenerationEU

Concurrent Setting

HAR.S.H.
HARdware-aware Similarity search
Greece 2.0
NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union
NextGenerationEU

Model

- The system is asynchronous.
- Processes communicate by accessing shared variables (base objects).
- In addition to atomic Read and Write, a process may execute an atomic CAS on a shared variable.




```
ATOMIC boolean CAS(Variable V,  Value v_old,
                                Value v_new) {
    if (V == v_old) {
        V = v_new;
        return TRUE;
    }
    return FALSE;
}
```

Locks

A **lock** is a programming construct that allows only one process to access a code section (called critical section) at each point in time.

Mutual Exclusion: Only one process can hold the lock at any given time.
Blocking: If a process attempts to acquire a lock that is already held, it is typically blocked (i.e., paused) until the lock is released.

No Deadlock: If some process has an active lock invocation at some point, then there is a later point at which SOME process has acquired the lock.
No starvation: If some process has an active lock invocation at some point, then there is a later point at which THAT SAME process has acquired the lock.

Interface of Locks

- **lock(L)**: is called for acquiring the lock L. If L is already taken, the process blocks until lock is released.
- **unlock(L)**: releases L so that other competing processes can acquire it.

```
while (TRUE) {
    lock(L)
    critical section
    unlock(L)
    reminder section
}
```

Implementation of Sets

- Use base objects to store the state of the set (i.e., the state of the linked list).
- Provide an algorithm for each process to implement each of the operations of the set.

Linearizability

- In each execution α, each operation should have the same response as if it has been executed serially (or atomically) at some point in its execution interval.
- This point is called linearization point of the operation.

- An implementation is linearizable if all the executions it produces are linearizable.

Progress

Wait-Freedom

- Each (non-faulty) process finishes the execution of its operation within a finite number of steps.

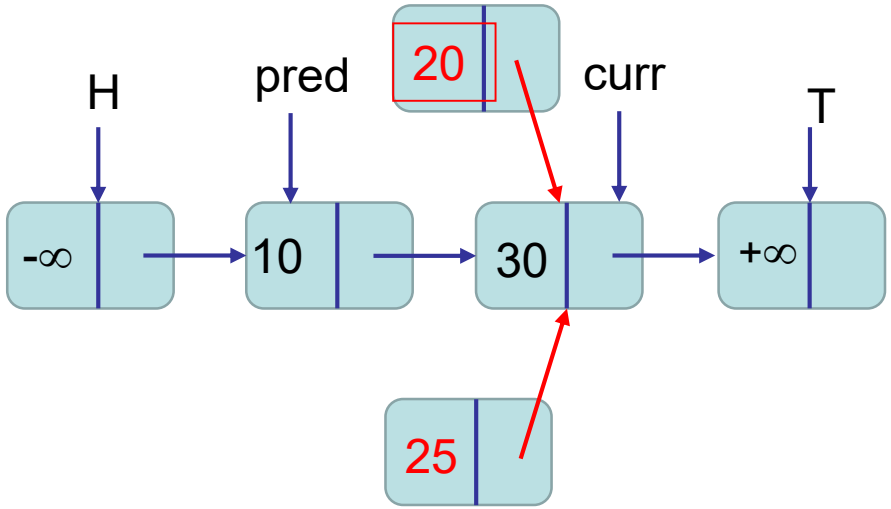
Lock-freedom

- If all non-faulty process continue to take steps, SOME process finishes the execution of its operation within a finite number of steps.

Blocking Algorithm

- A process may have to wait/block until another process takes some action.

Synchronization Problems when Updating Shared Lists

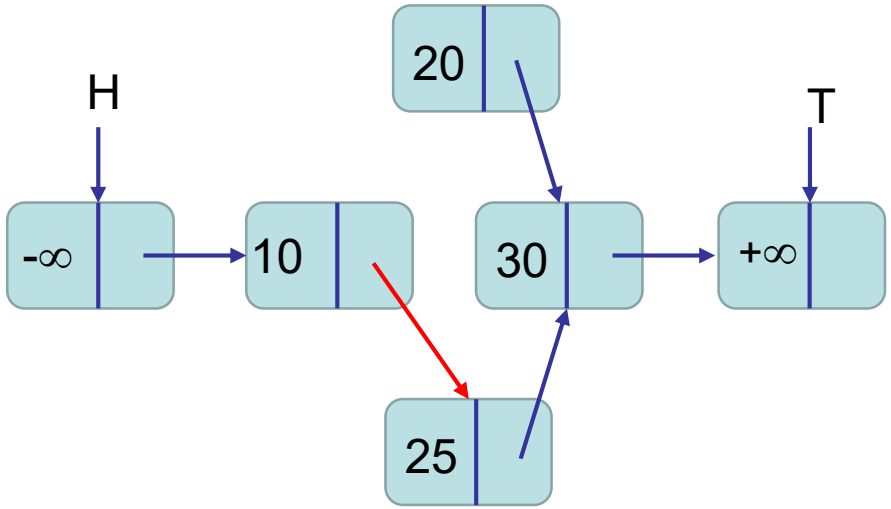


Concurrent insertions of nodes with keys 20 and 25

p1: Insert(20)

p2: Insert(25)

Synchronization Problems when Updating Shared Lists

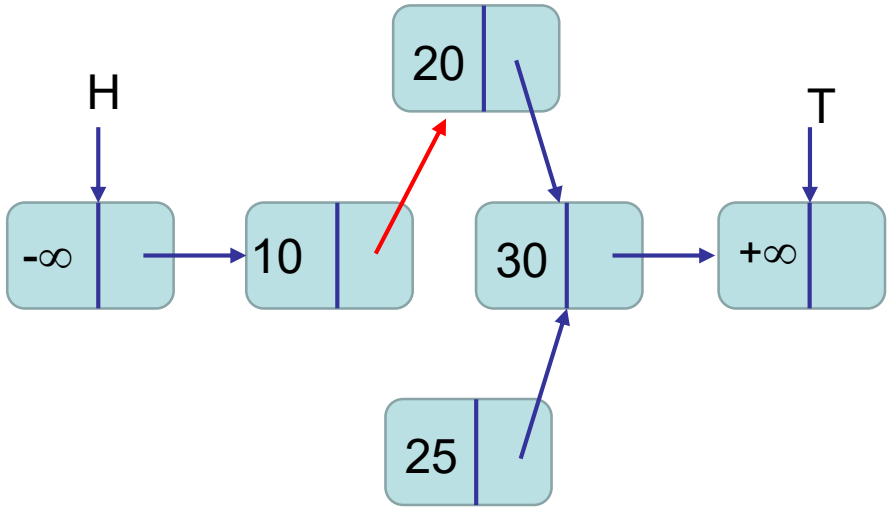


Concurrent insertions of nodes with keys 20 and 25

p1: Insert(20)

p2: Insert(25)

Synchronization Problems when Updating Shared Lists

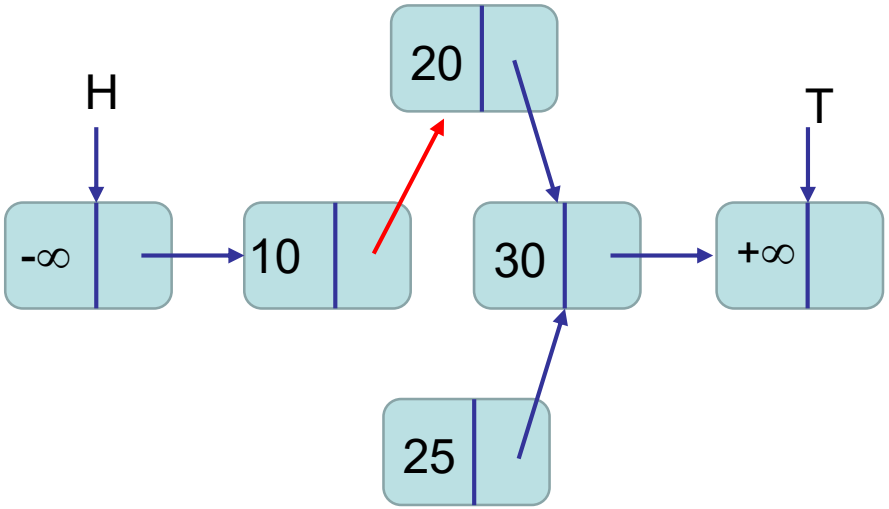


Concurrent insertions of nodes with keys 20 and 25

p1: Insert(20)

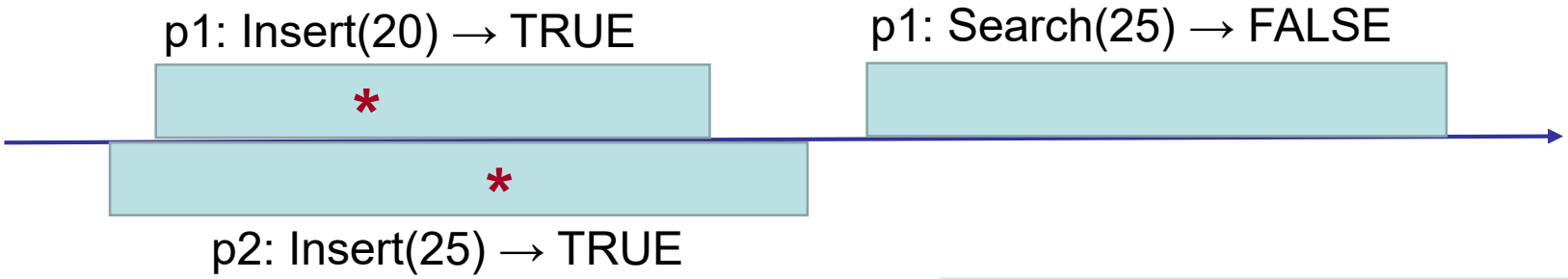
p2: Insert(25)

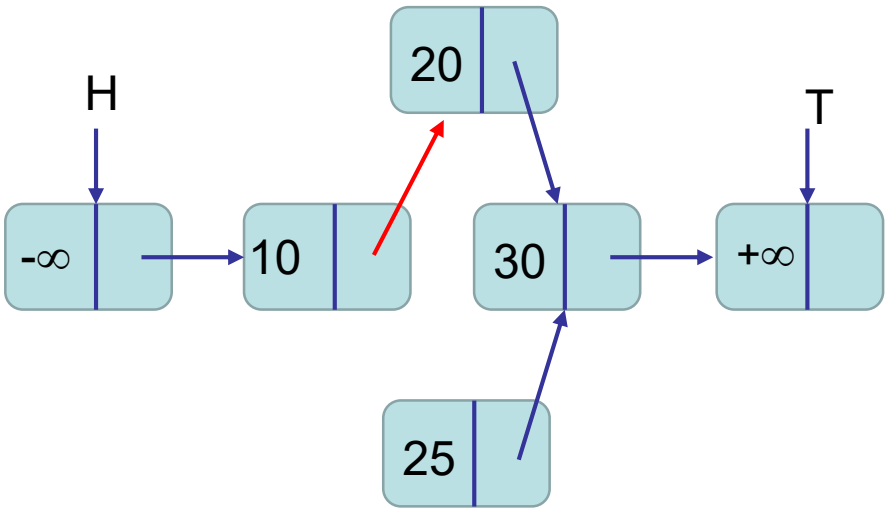
Synchronization Problems when Updating Shared Lists



If we traverse the list starting from H, key 25 is not in the list.

Why is this a problem?






```
boolean search(int key) {
    NODE *curr; boolean result;

   2. curr = H;
   3. while (curr->key < key)
   4.       curr = curr->next;
   5. if (key == curr->key)
   6.         result = TRUE;
   7. else result = FALSE;

   9. return result;
}
```

Assignment 1

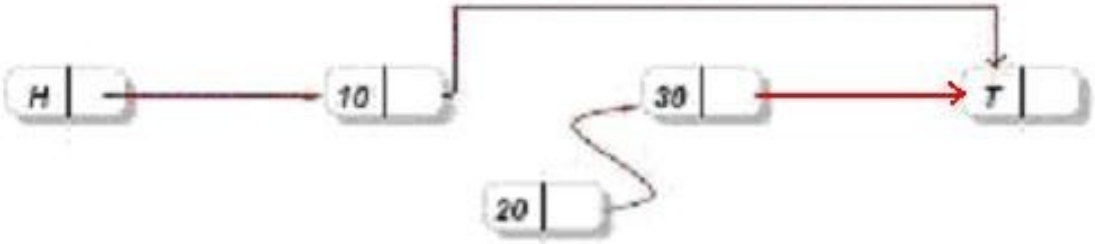
Can you come up with more bad scenarios? You may consider the following cases:

1. An Insertion that is executed concurrently with a Deletion
2. A Deletion that is executed concurrently with another Deletion.

Can you come up with a bad scenario, assuming that we use CAS (instead of Write) to update pointers, using as old value the value of curr.

HAR.S.H. HARdware-aware Similarity search
Greece 2.0 NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union NextGenerationEU

Synchronization Problems when Updating Shared Lists



Insertion of node with key 20 and concurrent deletion of node with key 30 in the list

Concurrent deletion of nodes with keys 10 and 30 from the list

Lock-Based Approaches for Implementing Concurrent Data Structures

1. Coarse-Grain Synchronization
2. Fine-grain Synchronization
3. Optimistic Synchronization
4. Lazy Synchronization

HAR.S.H. HARdware-aware Similarity search
Greece 2.0 NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union NextGenerationEU

Coarse-Grain Synchronization

HAR.S.H. HARdware-aware Similarity search
Greece 2.0 NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union NextGenerationEU

Linked Lists: Coarse-Grain Synchronization

```
boolean search(int key) {
     NODE *curr; boolean result;

    1. lock(L);
    2. curr = H;
    3. while (curr->key < key)
    4.    curr = curr->next;
    5. if (key == curr->key)
    6.         result = TRUE;
    7. else result = FALSE;
    8. unlock(L);
    9. return result;
}
```

```
typedef struct node {
            int key;
            NODE *next;
} NODE;

shared Lock L;
shared NODE *H, *T;
```

Linked Lists: Coarse-Grain Synchronization

```
boolean insert(int key, T x) {
// code for process p
    Node *pred, *curr;
    boolean result;

    10. lock(L);
    11. pred = H;
    12. curr = pred->next;
    13. while (curr->key < key) {
    14.       pred = curr;
    15.       curr = curr->next;
    }
    16. if (key == curr->key) result = FALSE;
    17. else {
    18.     NODE *node = newcell(NODE);
    19.     node->next = curr;
    20.     node->key = key;
    21.     pred->next = node;
     22.    result = TRUE;
    }
    23. unlock(L);
    24. return result;
}
```

```
boolean delete(int key) {
    // code for process p
    Node *pred, *curr;
    boolean result;

    25. lock(L);
    26. pred = H;
    27. curr = pred->next;
    28. while (curr->key < key) {
    29.       pred = curr;
    30.       curr = curr->next;
     }
    31. if (key == curr->key) {
    32.       pred->next = curr->next;
    33.       result = TRUE;
     }
    34. else result = FALSE;
    35. unlock(L);
    36.  return result;
}
```

Coarse-Grain Synchronization - Linearizability

- The algorithm is linearizable.

What do I have to do to prove it?

- Assign linearization points.
- Prove that linearization points are within execution intervals of operations.
- Prove responses of operations are consistent: operations in concurrent execution have the same responses as corresponding operations in sequential execution defined by linearization points.

Coarse-Grain Synchronization - Linearizability

How shall we assign linearization points to ops?

The linearization point for each operation can be placed at any point at which the operation has acquired the lock.

HAR.S.H. HARdware-aware Similarity search
Greece 2.0 NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union NextGenerationEU

Assignment 2

Consider the following assignment of linearization points:

- Delete and Search are linearized as described above.
- Insert is linearized when it returns (line 24).
- Present an execution that shows that this assignment of lin points is not correct.

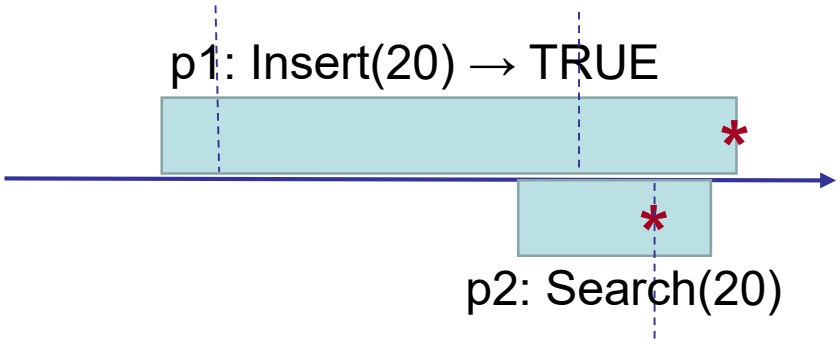
HAR.S.H. HARdware-aware Similarity search
Greece 2.0 NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union NextGenerationEU

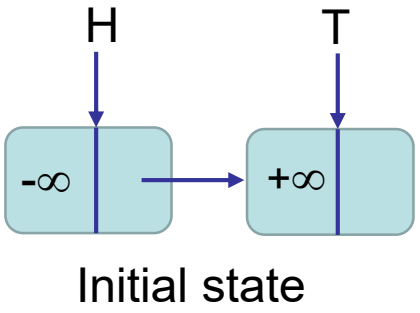
Linked Lists: Coarse-Grain Synchronization

```
boolean insert(int key, T x) {
// code for process p
    Node *pred, *curr;
    boolean result;

    10. lock(L);
    11. pred = H;
    12. curr = pred->next;
    13. while (curr->key < key) {
    14.       pred = curr;
    15.       curr = curr->next;
    }
    16. if (key == curr->key) result = FALSE;
    17. else {
    18.     NODE *node = newcell(NODE);
    19.     node->next = curr;
    20.     node->key = key;
    21.     pred->next = node;
     22.    result = TRUE;
    }
    23. unlock(L);
    24. return result;
}
```

```
boolean delete(int key) {
    // code for process p
    Node *pred, *curr;
    boolean result;

    25. lock(L);
    26. pred = H;
    27. curr = pred->next;
    28. while (curr->key < key) {
    29.       pred = curr;
    30.       curr = curr->next;
     }
    31. if (key == curr->key) {
    32.       pred->next = curr->next;
    33.       result = TRUE;
     }
    34. else result = FALSE;
    35. unlock(L);
    36.  return result;
}
```

Execution to illustrate Wrong Assignment of Lin Points





- What does p2 return in the concurrent execution above?
- What does it return in the sequential execution defined by the linearization points? [[[Search(20), Insert(20)]]]

Assignment 3

Consider the following assignment of linearization points:

- Insert and Search are linearized as described above.
- Delete is linearized when it is invoked (before executing line 25).
- Present an execution to illustrate that this assignment of lin points is not correct.

HAR.S.H. HARdware-aware Similarity search
Greece 2.0 NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union NextGenerationEU

Properties

- The implementation satisfies the same progress condition as the lock implementation it employs.
- If contention is low, the performance of the algorithm is ok.
- If contention is high, the algorithm performs poorly since parallelism is restricted.

Fine-Grain Synchronization

HAR.S.H. HARdware-aware Similarity search
Greece 2.0 NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union NextGenerationEU

Linked Lists: Fine-Grain Synchronization

- Each node is associated with its own lock.
- Locks are acquired in a hand-over-hand manner.
  - While holding the lock of the node pointed to by pred, acquire the lock of the node pointed to by curr, and then release the lock of the node pointed to by pred.
  - This is called hand-over-hand locking or lock coupling.
- To avoid deadlocks, the locks should be acquired in the same order by each process.

```
typedef struct node {
        int key; Lock lock; NODE *next;
} NODE;

boolean search(int key) {
    NODE *curr, *pred; boolean result;

    lock(H->lock);
    pred = H;
    curr = pred->next;
    lock(curr->lock);
    while (curr->key < key) {
            unlock(pred->lock);
            pred = curr;
            curr = curr->next;
            lock(curr->lock);
    }
    if (key == curr->key) result = TRUE;
    else result = FALSE;
    unlock(pred->lock); unlock(curr->lock);
    return result;
}
```

Linked Lists: Fine-Grain Synchronization

```
boolean insert(int key) { // code for process p
    Node *pred, *curr; boolean result;

    lock(H->lock);
    pred = H;
    curr = pred->next;
    lock(curr->lock);
    while (curr->key < key) {
            unlock(pred->lock);
            pred = curr;
            curr = curr->next;
            lock(curr->lock);
    }
    if (key == curr->key) result = FALSE;
    else {
        NODE *node = newcell(NODE);
        node->next = curr;
        node->key = key;
        pred->next = node;
        result = TRUE;
    }
    unlock(pred->lock);
    unlock(curr->lock);
    return result;
}
```

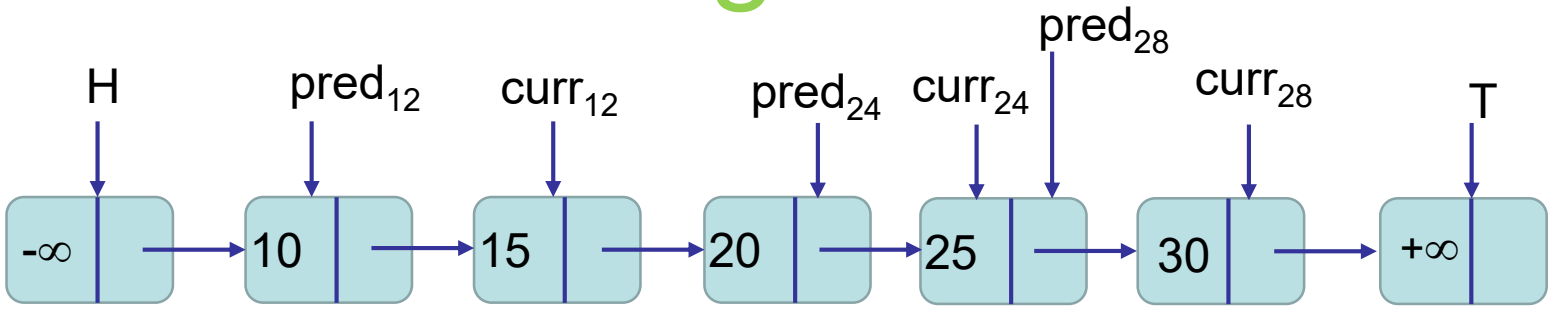
```
boolean delete(int key) {
    // code for process p
    Node *pred, *curr;
    boolean result;

    lock(head->lock);
     pred = H;
     curr = pred->next;
     lock(curr->lock);
     while (curr->key < key) {
            unlock(pred->lock);
            pred = curr;
            curr = curr->next;
            lock(curr->lock);
     }
     if (key == curr->key) {
            pred->next = curr->next;
            result = TRUE;
     }
     else result = FALSE;
     unlock(pred->lock);
     unlock(curr->lock);
     return result;
}
```

Assignment 3

- Does this algorithm ensures a higher degree of parallelism in comparison to coarse-grain lock?
- Does it result in the best degree of parallelism?
- Can you come up with a scenario that parallelism is restricted?
- Does this algorithm results in best of performance?

HAR.S.H. HARdware-aware Similarity search
Greece 2.0 NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union NextGenerationEU

Assignment



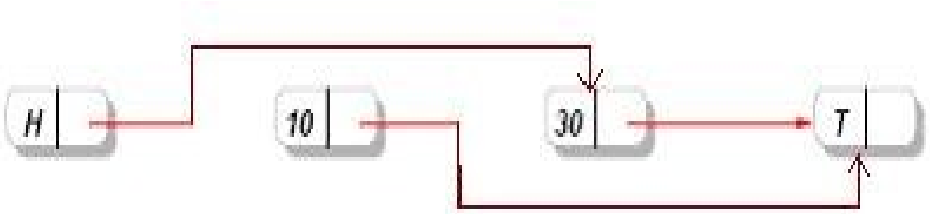
Consider the following operations:

- Insert(12)
- Insert(24)
- Insert(28)

Can they all proceed concurrently?

Does the fine-grain implementation allows to all these operations to proceed concurrently?

Assignment



Why hand-over-hand locking is required?
Why delete() must acquire two locks?
What might go wrong if we acquire just the lock of Node pointed to by curr?
What might go wrong if we acquire just the lock of Node pointed to by pred?

Fine Grain Synchronization

Linearizability

How shall we assign linearization points?

- A successful insert(k, x) is linearized when the node with the next higher key is locked.
- An unsuccessful insert(k, x) is linearized when the node with key k is locked.
- Similar rules apply for delete.
- Linearization points for search -> when the node with key k (if any) or with the next higher key (if not) is locked.

Fine-Grain Synchronization - Questions

Progress

- Is the algorithm starvation-free?

- Yes, assuming that all individual locks are starvation-free.

- Is deadlock possible?
- No
  - If a process p attempts to lock head, it eventually succeeds.
  - Eventually, all locks held by other processes will be released and p will manage to lock $pred_p$ and $curr_p$.

HAR.S.H. HARdware-aware Similarity search
Greece 2.0 NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union NextGenerationEU

Optimistic Synchronization

HAR.S.H. HARdware-aware Similarity search
Greece 2.0 NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union NextGenerationEU

Linked Lists – Optimistic Synchronization

Main Ideas

- **Search without taking into consideration locks**
- Lock the found nodes (pred and curr)
- Confirm that the locked nodes are correct.
  - Use some form of validation.

```
boolean validate(NODE *pred, NODE *curr)
{
   NODE *tmp = H;

   while (tmp->key ≤ pred->key) {
      if (tmp == pred)  {
          if (pred->next == curr) return TRUE;
          else return FALSE;
      }
      tmp = tmp->next;
   }
   return FALSE;
}
```

```
boolean search(int key) {
    NODE *curr; boolean result;

    while (TRUE) {
        pred = H; curr = pred->next;
        while (curr->key < key) {
            pred = curr; curr = curr->next;
        }
        lock(pred->lock); lock(curr->lock);
        if (validate(pred, curr) == TRUE) {
            if (key == curr->key) result = TRUE;
            else result = FALSE;
            return_flag = 1;
        }
        unlock(pred->lock); unlock(curr->lock);
        if (return_flag) return result;
    }
}
```

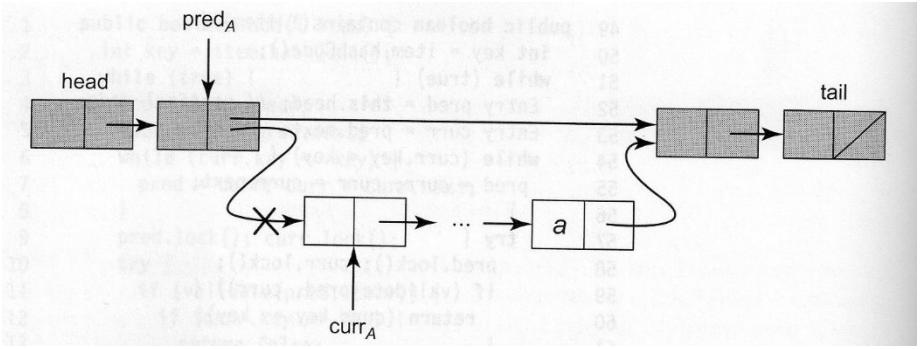
Linked Lists – Optimistic Synchronization

```
boolean insert(int key, T x) { // code for process p
    Node *pred, *curr;
    boolean result;
    boolean return_flag = 0;

    while (TRUE) {
        pred = head;   curr = pred->next;
        while (curr->key < key) {
                pred = curr;
                curr = curr->next;
        }
        lock(pred->lock); lock(curr->lock);
        if (validate(pred, curr) == TRUE) {
            if (key == curr->key) {
                result = FALSE; return_flag = 1;
            }
            else {
                NODE *node = newcell(NODE);
                node->next = curr;
                node->key = key;
                pred->next = node;
                result = TRUE; return_fl ag = 1;
            }
        }
        unlock(pred->lock); unlock(curr->lock);
        if return_flag) return result;
    }
}
```

```
boolean delete(int key) {
    // code for process p
    Node *pred, *curr;
    boolean result;
    boolean return_flag = 0;

    while (TRUE) {
        pred = head;  curr = pred->next;
        while (curr->key < key) {
                pred = curr;
                curr = curr->next;
        }
        lock(pred->lock); lock(curr->lock);
        if (validate(pred, curr)) {
            if (key == curr->key) {
                    pred->next = curr->next;
                    result = TRUE;
                }
            else result = FALSE;
            return_flag = 1;
        }
        unlock(pred->lock); unlock(curr->lock);
        if (return_flag == 1) return result;
    } }
```

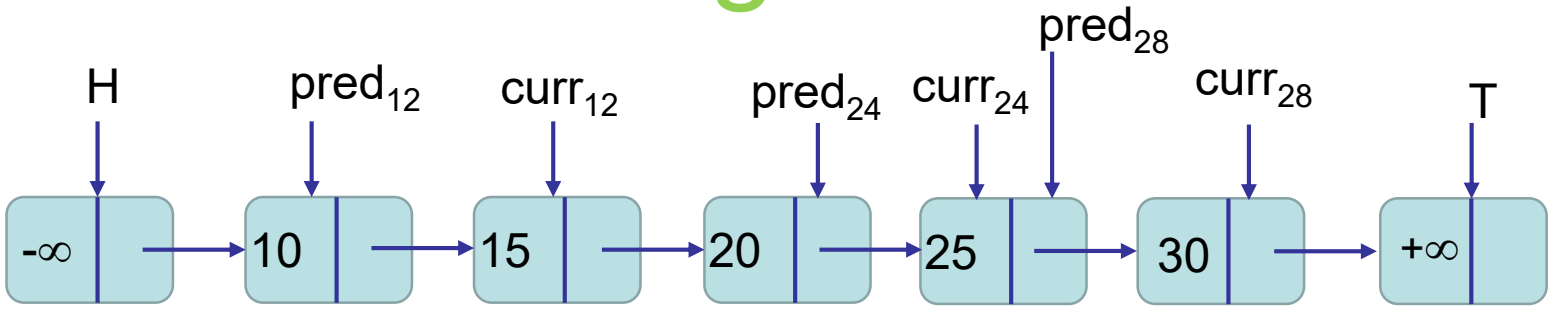
Linked Lists – Optimistic Synchronization Assignment

M. Herlihy and N. Shavit, The Art of Multiprocessor Programming



1. Is validation necessary?
2. Why? Can you come up with a scenario in which an operation traverses nodes that have been deleted from the list?
3. What will happen if a process follows the next fields of deleted nodes? Will it eventually return to some node of the list?

Assignment



Consider the following operations:

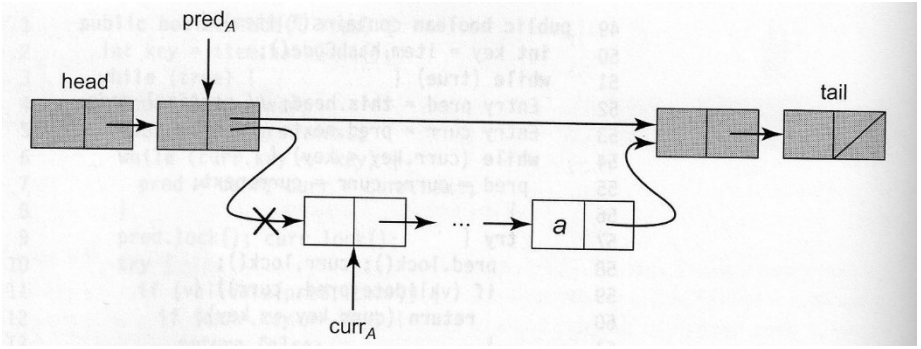
- Insert(12)
- Insert(24)
- Insert(28)

Can they all proceed concurrently?

Does the optimistic implementation allows these operations to proceed concurrently?

Optimistic Synchronization Discussion Point

M. Herlihy and N. Shavit, The Art of Multiprocessor Programming



How shall we assign linearization points?

Optimistic Synchronization Some properties

What are the progress guarantees provided by the algorithm?

- The algorithm is not starvation-free, even if all nodes' locks are starvation-free. Why? Bad scenario where all processes starve?
- What do you have to say in terms of performance?
- The implementation works well if the cost of traversing the list twice without locking is significantly less than the cost of traversing the list once with locking.

Lazy Synchronization

HAR.S.H. HARdware-aware Similarity search
Greece 2.0 NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union NextGenerationEU

Linked Lists – Lazy Synchronization

Main Ideas

- We add in each node a boolean marked field which indicates whether this node is in the set.
- **Traversals do not lock and do not validate.**
- Insert() locks the target's predecessor and adds the new node.
- Delete is realized in two steps:
  - mark the node as deleted
  - physically remove the node

```
boolean search(int key) {
    NODE *curr;
    boolean result;

    curr = head;
    while (curr->key < key)
        curr = curr->next;
    if (curr->marked !=TRUE
        && key==curr->key)
        return TRUE;
    else return FALSE;
}
```

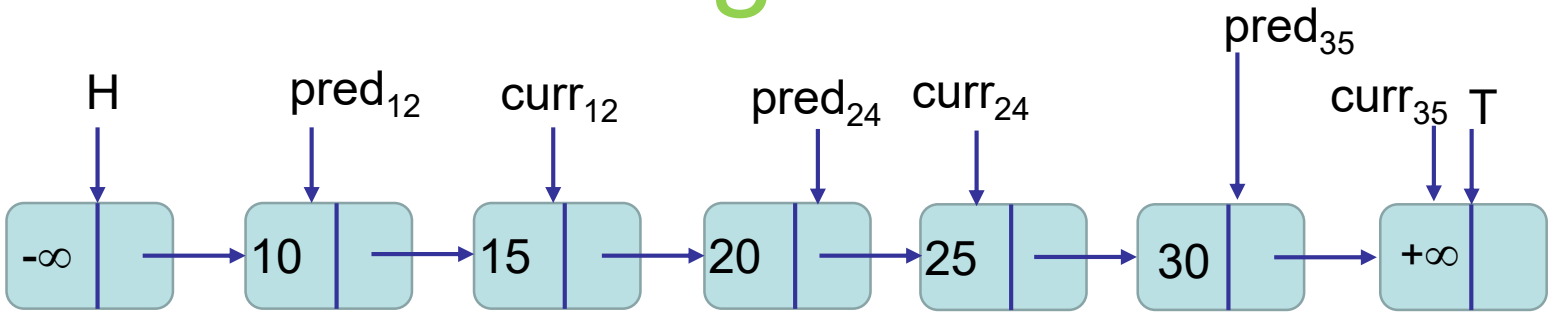
```
boolean validate(NODE *pred, NODE *curr) {
    if (pred->marked == FALSE &&
        curr->marked === FALSE &&
        pred->next == curr) return TRUE;
    else return FALSE;

}
```

Linked Lists – Lazy Synchronization

```
boolean insert(int key, T x) {      // code for process p
    Node *pred, *curr;
    boolean result;
    boolean return_flag = 0;

    while (TRUE) {
        pred = H;   curr = pred->next;
        while (curr->key < key) {
                pred = curr;
                curr = curr->next;
        }
        lock(pred->lock); lock(curr->lock);
        if (validate(pred, curr) == TRUE) {
            if (key == curr->key) {
                result = FALSE; return_flag = 1;
            }
            else {
                NODE *node = newcell(NODE);
                node->next = curr;
                node->key = key;
                pred->next = node;
                result = TRUE; return_flag = 1;
            }
        }
        unlock(pred->lock); unlock(curr->lock);
        if return_flag) return result;
    }
}
```

```
boolean delete(int key) {
    // code for process p
    Node *pred, *curr;
    boolean result; boolean return_flag = 0;

    while (TRUE) {
        pred = H;  curr = pred->next;
        while (curr->key < key) {
                pred = curr;
                curr = curr->next;
        }
        lock(pred->lock); lock(curr->lock);
        if (validate(pred, curr)) {
            if (key == curr->key) {
                    curr->marked = TRUE;
                    pred->next = curr->next;
                    result = TRUE;
                }
            else result = FALSE;
            return_flag = 1;
        }
        unlock(pred->lock); unlock(curr->lock);
        if (return_flag == 1) return result;
    }
}
```

Assignment



Consider the following operations:

- Insert(12)
- Insert(24)
- Insert(35)

Can they all proceed concurrently?

Does the lazy synchronization technique allows all of these operations to proceed concurrently?

Linked Lists – Lazy Synchronization

Progress?

- Insert() and Delete() are NOT starvation-free since list traversals may be arbitrarily delayed by ongoing modifications.

- Can you come up with a scenario where a process starves?

HAR.S.H. HARdware-aware Similarity search
Greece 2.0 NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union NextGenerationEU

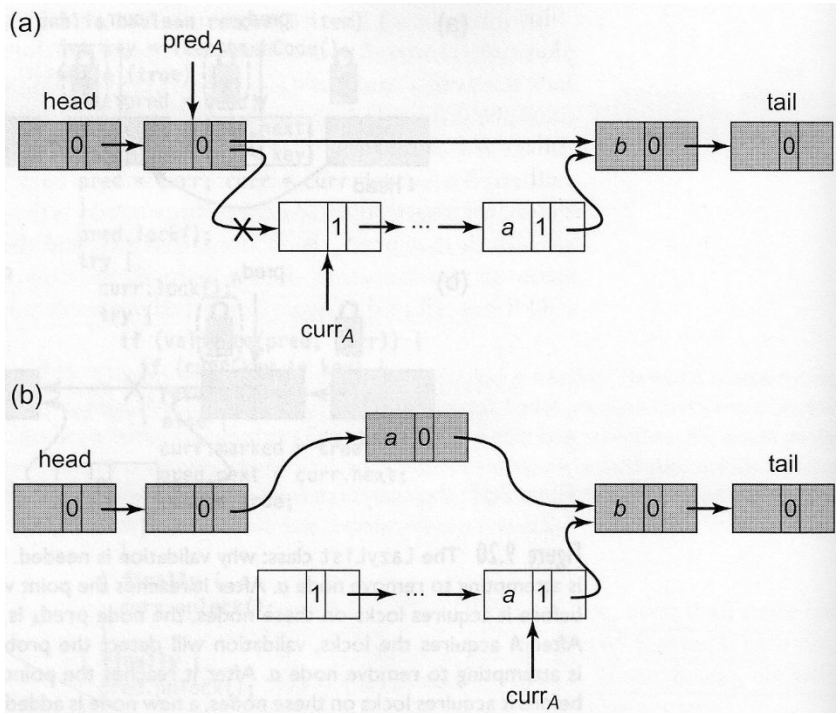
Linked Lists – Lazy Synchronization

Linearization Points

- **Insert()?**
  - Successful: when pred->next changes to point to node.
  - Unsuccessful: at the point that it acquires the lock to curr for the last time.
- **Delete()?**
  - Successful: when the mark is set.
  - Unsuccessful: at the point that it acquires the lock to curr for the last time.
- **Search()?**
  - Successful: when an unmarked matching node is found.
  - Unsuccessful: Can we linearize an unsuccessful search when it detects that the node it is looking for is marked?

Linked Lists – Lazy Synchronization

M. Herlihy and N. Shavit, The Art of Multiprocessor Programming



An unsuccessful search() is linearized at the earlier of the following points:

1. the point where a removed matching node, or a node with key greater than the one being searched is found, and
2. the point immediately before a new matching node is inserted to the list.

Bibliography

These slides are based on material that appears in the following book:

- M. Herlihy and N. Shavit, The Art of Multiprocessor Programming, Morgan Kauffman, 2008

HAR.S.H. HARdware-aware Similarity search
Greece 2.0 NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union NextGenerationEU

Concurrent Tree-Based Dictionaries

HAR.S.H. HARdware-aware Similarity search
Greece 2.0 NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union NextGenerationEU

Efficient Concurrent Implementations of Binary-Search Trees

- A lock-free implementation of BSTs from single-word CAS.

Properties

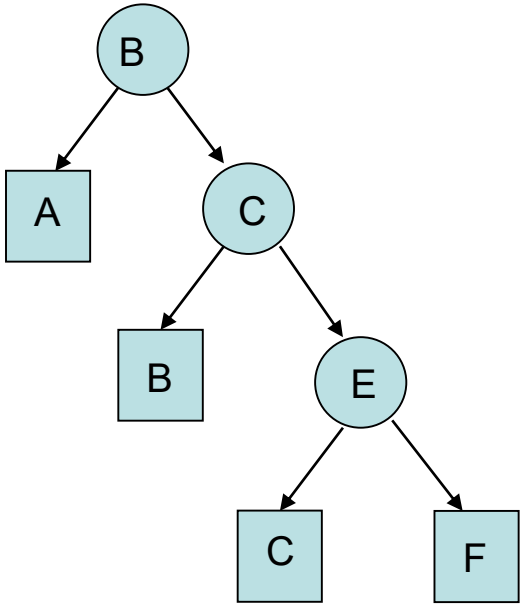
- Conceptually simple
- Allows for fast searches
- Concurrent updates to different parts of the tree do not conflict
- Experiments show good performance

HAR.S.H. HARdware-aware Similarity search
Greece 2.0 NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union NextGenerationEU

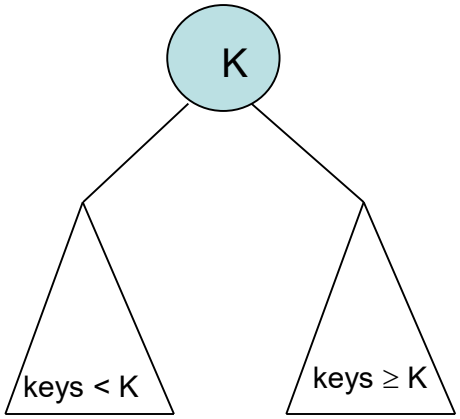
Leaf-Oriented BST

Properties

- One leaf for each key in set
- Internal nodes used for routing
- The tree is full
- BST Property



Leaf-Oriented BST storing key set {A,B,C,F}



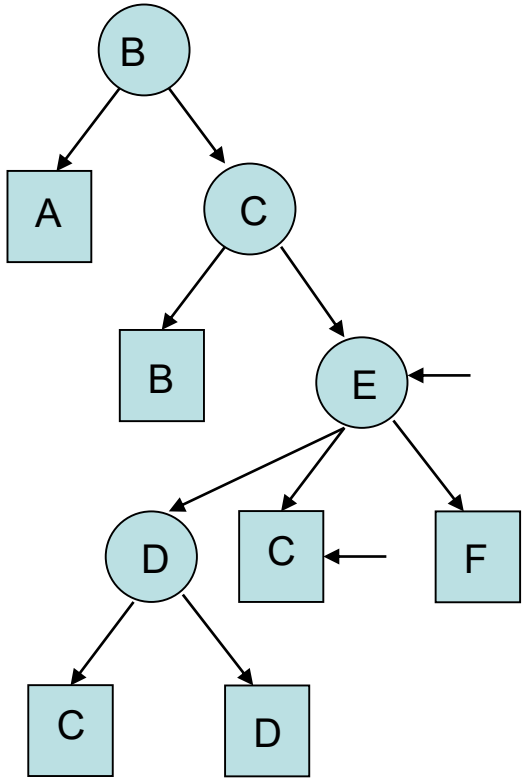
Why Leaf-Oriented BSTs?

- Deletions are simpler.
- Average depth is only slightly higher.

Insertion (sequential version)

Insert(D)

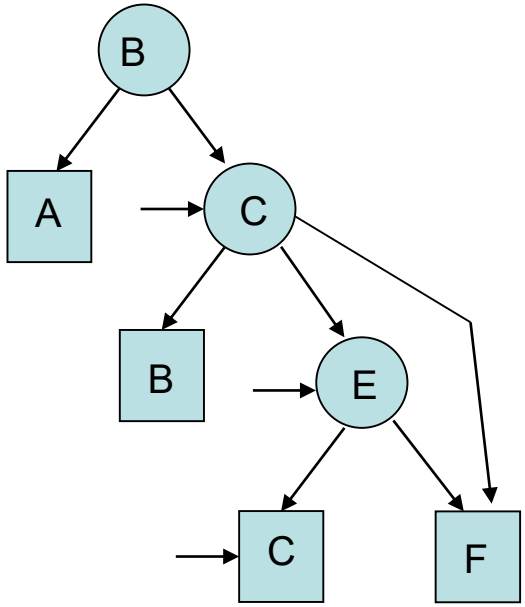
- Search for D
- Remember leaf and its parent
- Create new leaf, replacement leaf and one internal node
- Update pointer



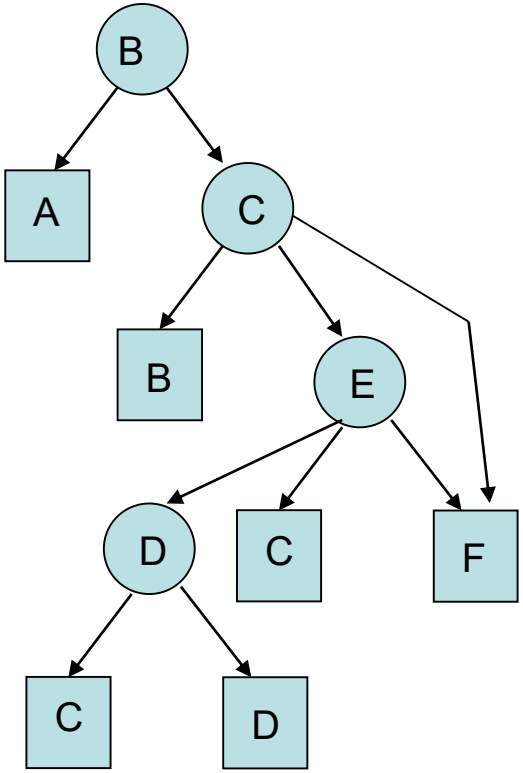
Deletion (sequential version)

Delete(C)

- Search for C
- Remember leaf, its parent and grandparent
- Update pointer



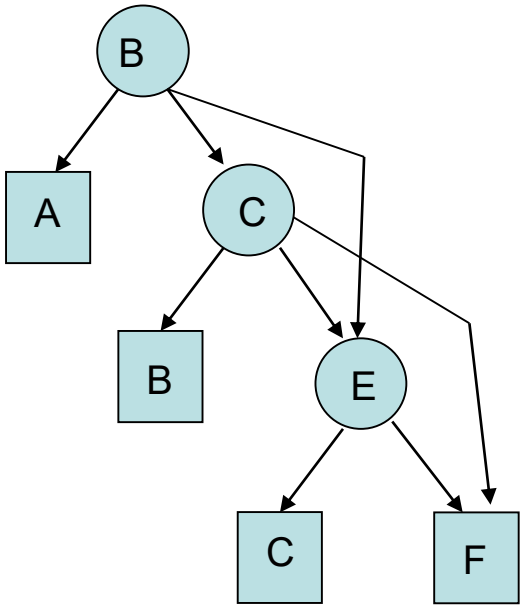
Challenges of Concurrency



Concurrently, Delete(C) and Insert(D)

- D is not reachable from the root!

Challenges of Concurrency



Concurrent Deletion of B and C

- C is still reachable from the root!

Problem

- A node's child pointer is changed while the node is being removed from the tree!

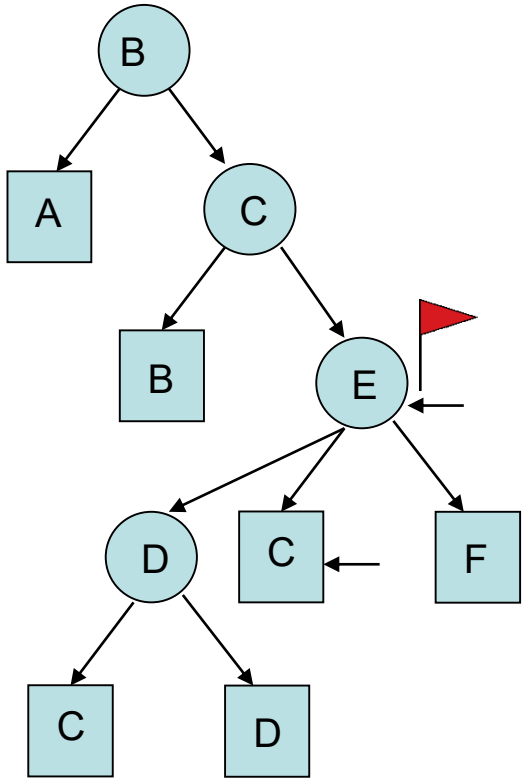
NB-BST: The concurrent BST Algorithm

- Flags and marks internal nodes during updates.
- Flag: indicates that an update is changing a child pointer
  - Flag an internal node x before changing a child pointer of it
  - Once the update has been performed, unflag x
- Mark: indicates that an internal node has be or soon will be removed from the tree
  - Mark an internal node x before removing it
  - Node remains marked forever

The Insertion Algorithm

Insert(D)

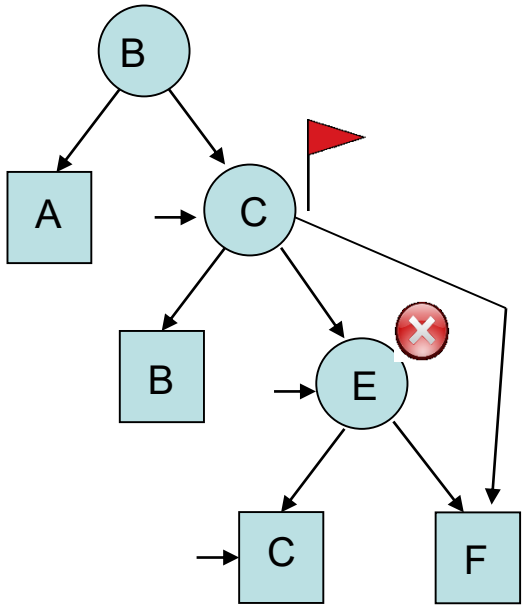
- Search for D
- Remember leaf and its parent
- Create three new nodes
- Flag parent (if this fails, step back and retry)
- Update pointer (using CAS)
- Unflag parent



The Deletion Algorithm

Delete(C)

- Search for C
- Remember leaf, its parent and grandparent
- Flag grandparent (if this fails, step back and retry)
- Mark parent (if this fails, unflag grandparent, step back, and retry)
- Update pointer
- Unflag grandparent



Conflicting Operations now work

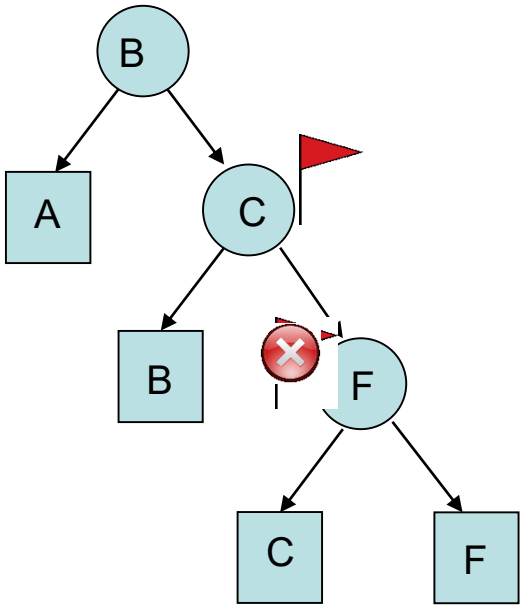
Concurrently, Delete(C) and Insert(D)

Case 1: Delete(C)'s flag and mark succeeds

- Delete(C) will complete
- Insert(D)'s flag fails
  - Insert(D) will step back and retry

Case 2: Insert(D)'s flag succeeds

- Insert(D) will complete
- Delete(C)'s mark fails
  - Delete(C) will step back and retry

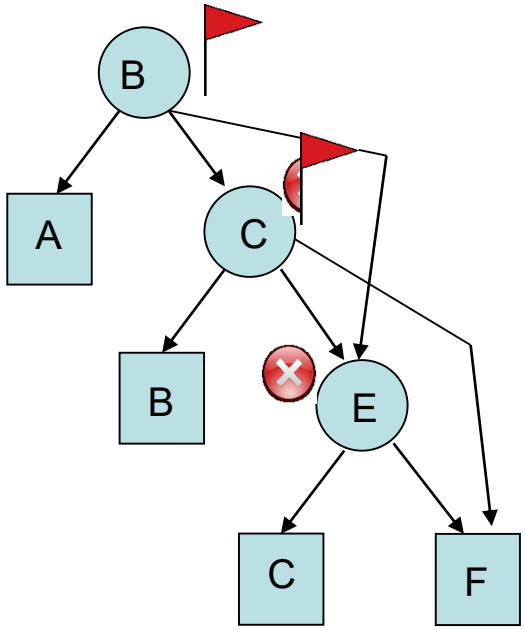


Conflicting Operations now Work!

Concurrent Deletion of B and C

Case 1: Delete(B)'s flag and mark succeeds

- Delete(B) will complete
- Delete(C)'s flag fails
  - Delete(C) has to step back and retry

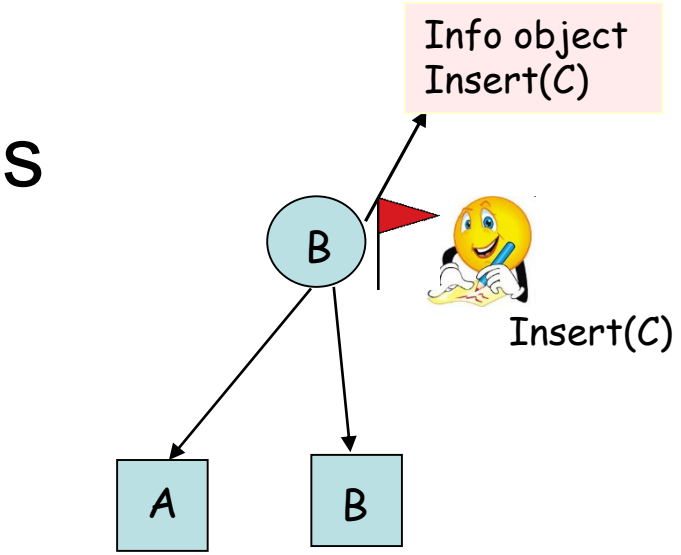


Case 2: Delete(C)'s flag succeeds

- Delete(C) will complete
- Delete(B)'s mark fails
  - Delete(B) has to step back and retry

Lock-Freedom

- Whenever flagging or marking, leave a key under the doormat.
  - A flag or mark points to a small record that tells a thread how to help the original operation.
- If an operation fails to flag or mark, it helps the previous operation complete before retrying.



The algorithm for Find

- Searches just traverse a path of the BST until reaching a leaf.
- They ignore flags and marks.

➢ **Algorithm for Find is similar to its sequential version**

HAR.S.H. HARdware-aware Similarity search
Greece 2.0 NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union NextGenerationEU

Thank you!

https://harsh-project.gr/

https://persist-project.gr/

Current Projects
I am looking for promising young researchers to recruit!

faturu@csd.uoc.gr
www.ics.forth.gr/~faturu/

HAR.S.H. HARdware-aware Similarity searcH
Greece 2.0 NATIONAL RECOVERY AND RESILIENCE PLAN
Funded by the European Union NextGenerationEU
persist
H.F.R.I. Hellenic Foundation for Research & Innovation