Announcements

- HW2 due next Tuesday, Feb 6

Competitiveness of Locking

- Problem statement: Consider a process waiting for a lock. There are two choices for an action that it can take:
  - The process can **spin**, at cost proportional to the length of time it spins, or
  - The process can **block**, which has **context-switch cost** C (time to switch out and back).

- An online algorithm for the spin-block problem must decide how long to spin before it blocks.
  - The **deterministic algorithm** that spins equal to the context-switch time is 2-competitive with the optimal algorithm.
  - The **randomized algorithm** that picks a time to switch to blocking between 0 and C is e/(e-1) ~ 1.58-competitive.

From “Competitive Randomized Algorithms for Non-Uniform Problems” by Karlin et al.

Optimistic Concurrency Control in Linked Lists

Question: What happens if you are traversing the list and another thread deletes the elements out from under you?

Unlikely because deletes are not very common compared to insert/find

Solution from last time: **Reference counting** to avoid getting garbage collected so you can always get to the end. Even if you grab the locks, you have to start again because verification will fail.

Another idea: Combine it with **lazy synchronization**

Lazy removal

```
public boolean remove(int key) {
   while (true) {
     Entry pred = this.head;
     Entry curr = head.next;
     while (curr.key < key) {
       pred = curr; curr = curr.next;
     }
     pred.lock();
     try {
       curr.lock();
       try {
         if (validate(pred, curr)) {
           if (curr.key != key) {     // present
             return false;
           } else {                   // absent
LR:          curr.marked = true;    // logically remove
             pred.next = curr.next;  // physically remove
             return true;
           }
         }
       } finally {                    // always unlock curr
         curr.unlock();
       }
     } finally {                      // always unlock pred
       pred.unlock();
     }
   }
 }
```

Logically remove element before physically removing it

From “A Lazy Concurrent List-Based Set Algorithm” - Heller et al. OPODIS 2005

Validation

```
private boolean validate(Entry pred, Entry curr) {
  return  !pred.marked && !curr.marked && pred.next == curr;
}
```

From “A Lazy Concurrent List-Based Set Algorithm” - Heller et al. OPODIS 2005

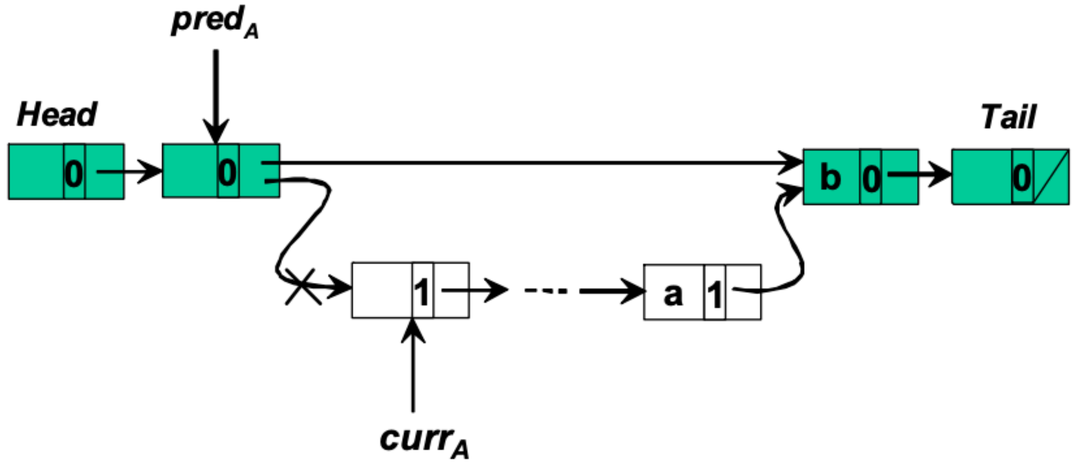
From “A Lazy Concurrent List-Based Set Algorithm” - Heller et al. OPODIS 2005

List traversal

- A concurrent physical removal of a node during thread A’s traversal can cause it to **traverse a physically removed part** of the list.

- The traversal works correctly assuming **freedom from interference**, which implies that nodes, even if they are removed from the list, are not recycled as long as they are reachable.

- Therefore, even if a node is removed while it is being traversed, the thread will keep following the pointers and **eventually reach the target**.



B+-tree Concurrency Control (continued)

Recall: Latch Crabbing / Coupling

Find: Start at root and traverse down to the correct leaf.

- Acquire **R**(eader) latch on child.
- Then unlatch parent.

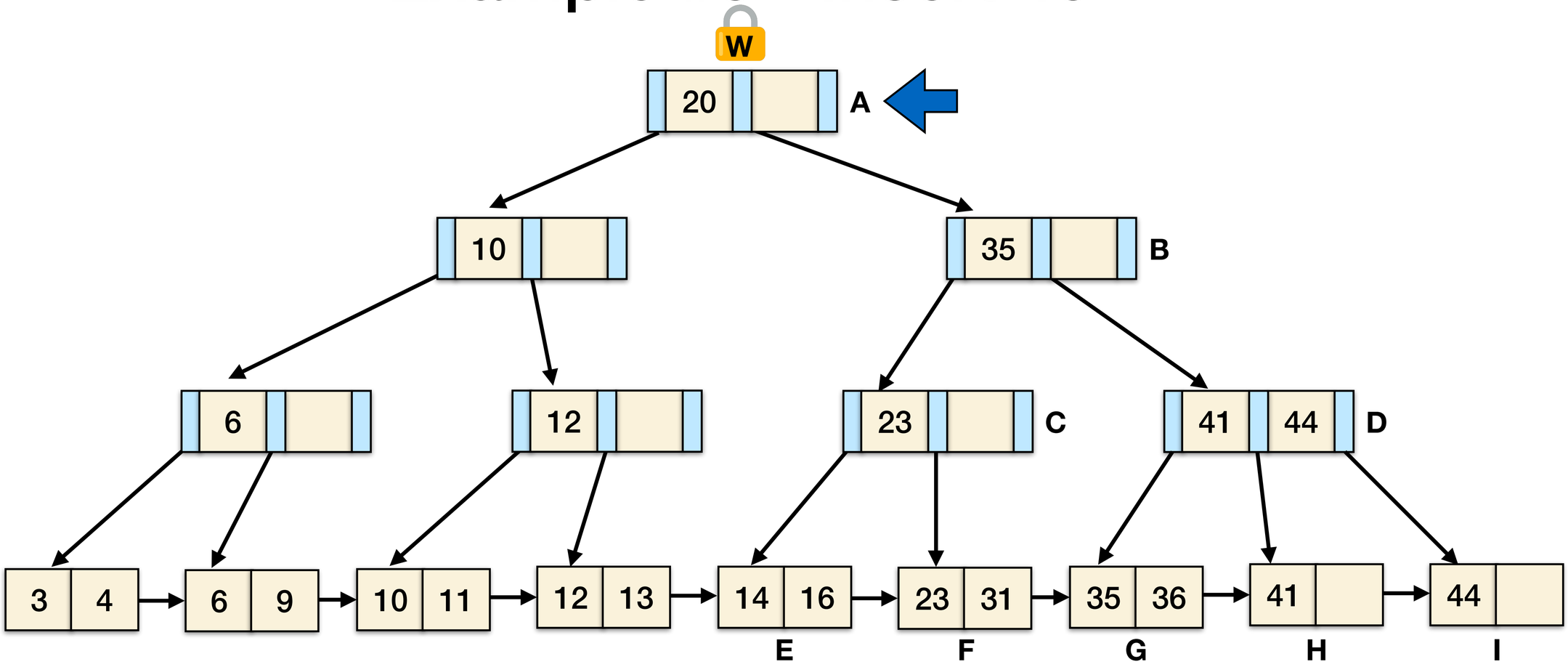
Similar to hand-over-hand

Insert/Delete: Start at root, and go down, obtaining **W**(riter) latches as needed.

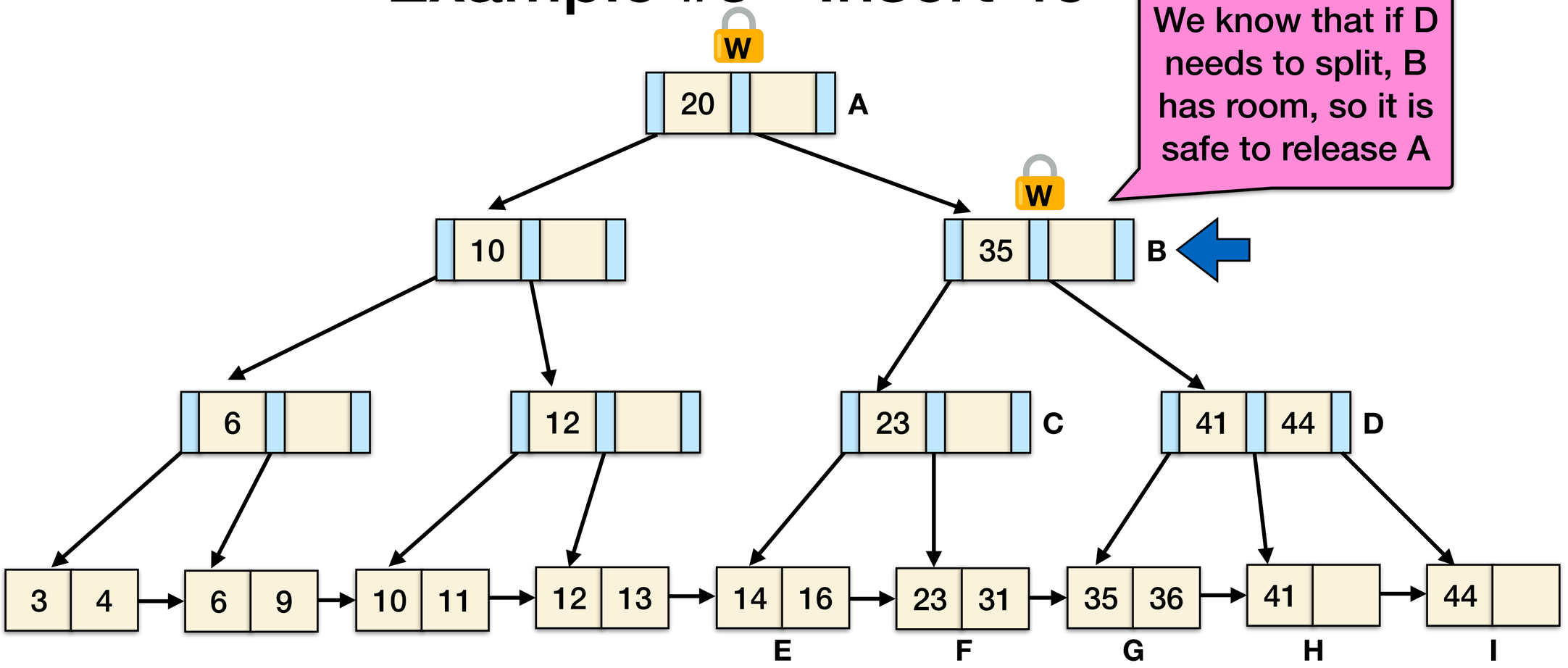
- Once the child is latched, check that it is safe.
- If it is safe, release all latches on ancestors.

Modified hand-over-hand: can keep hold on ancestors, if necessary

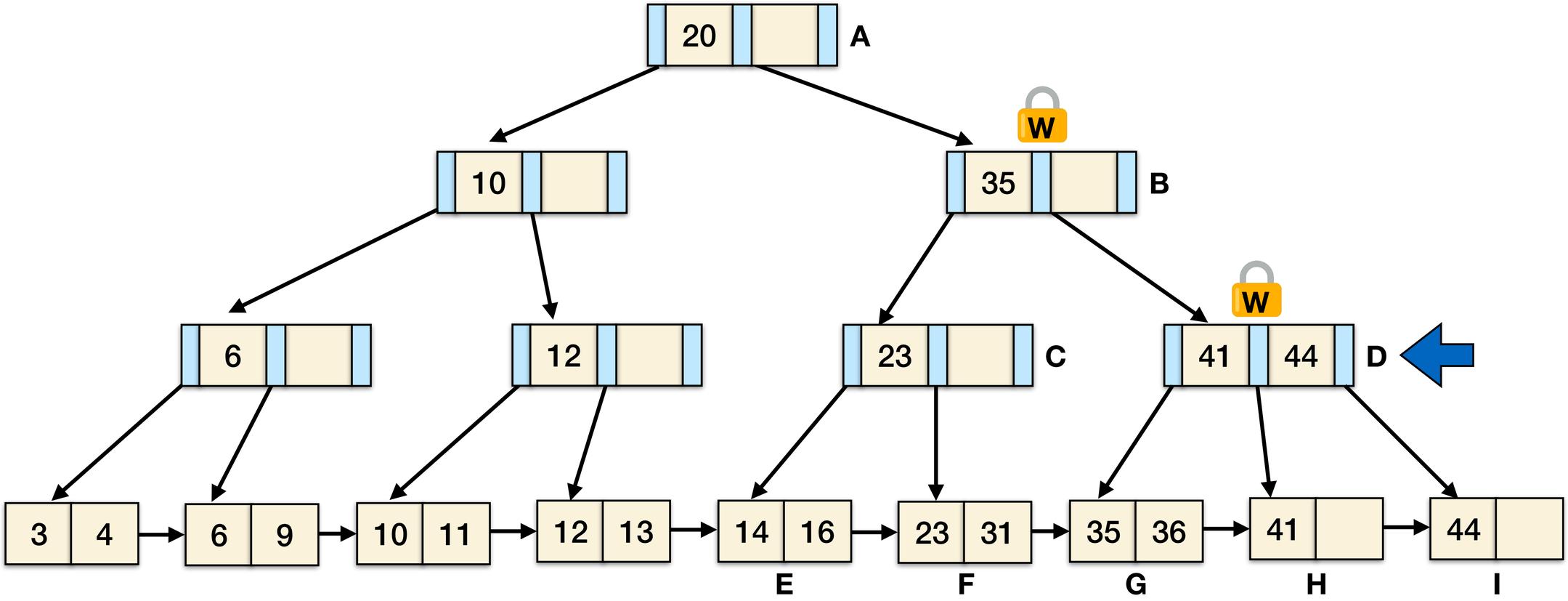
Example #3 - Insert 45



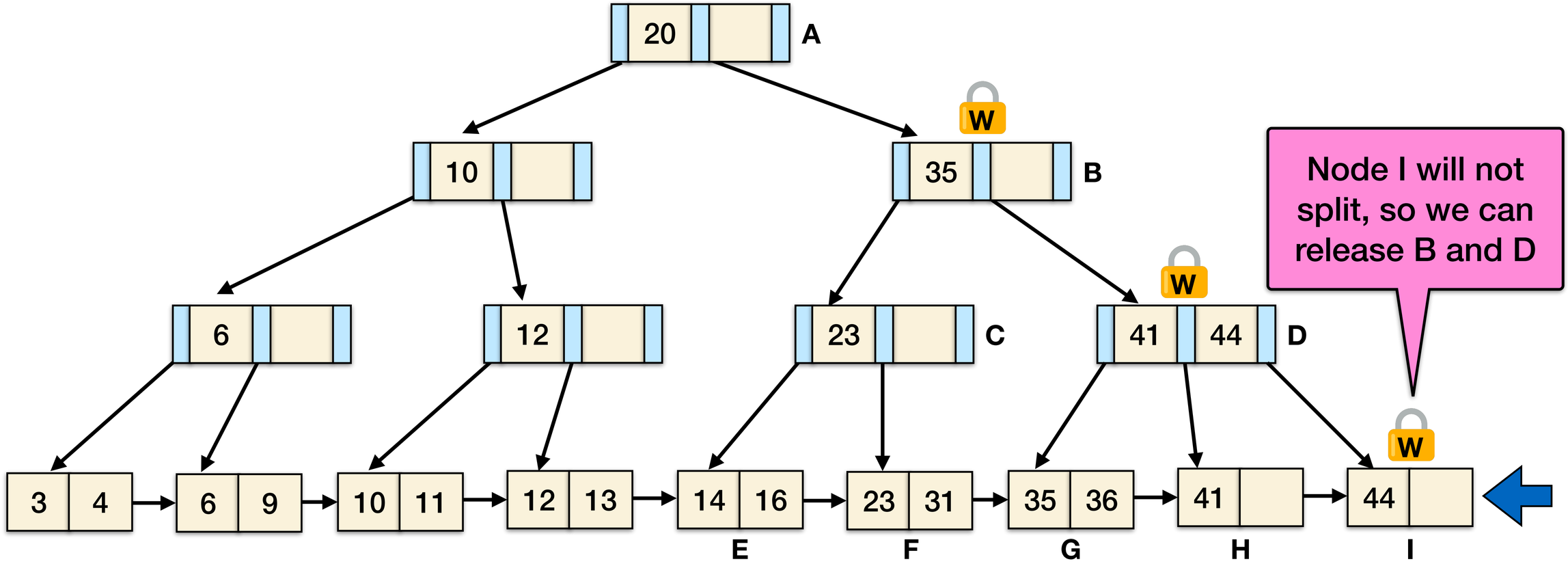
Example #3 - Insert 45



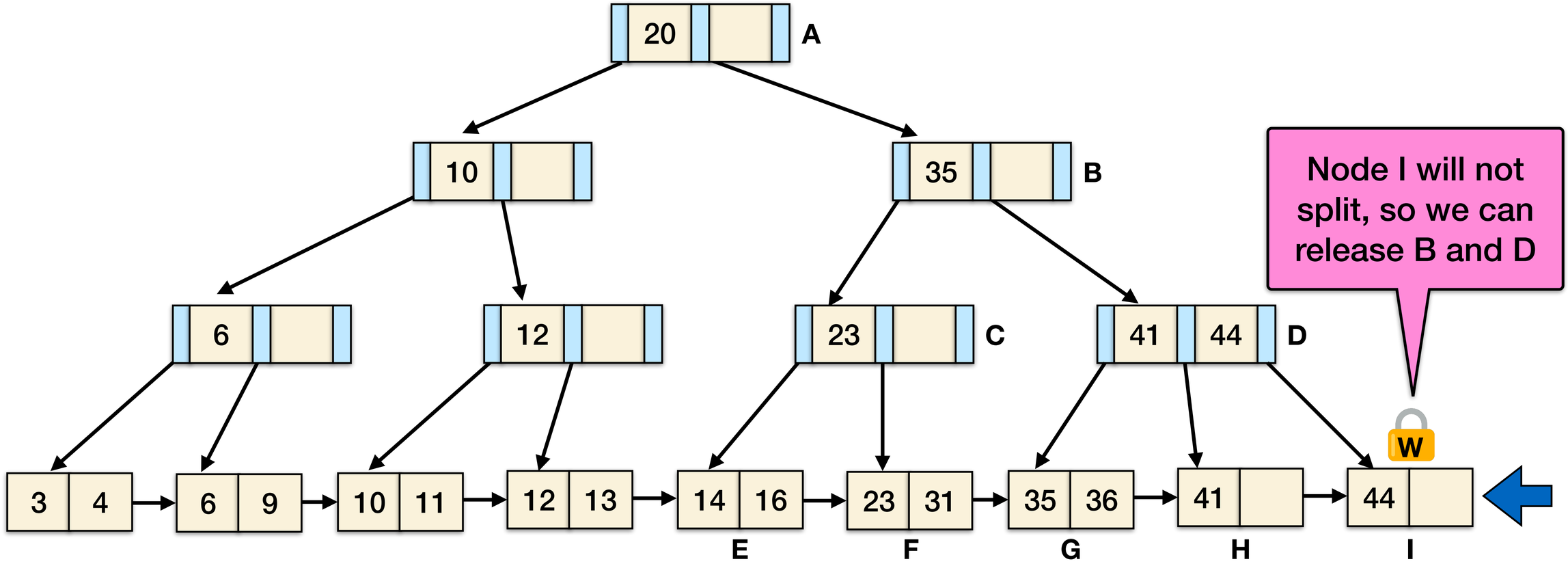
Example #3 - Insert 45



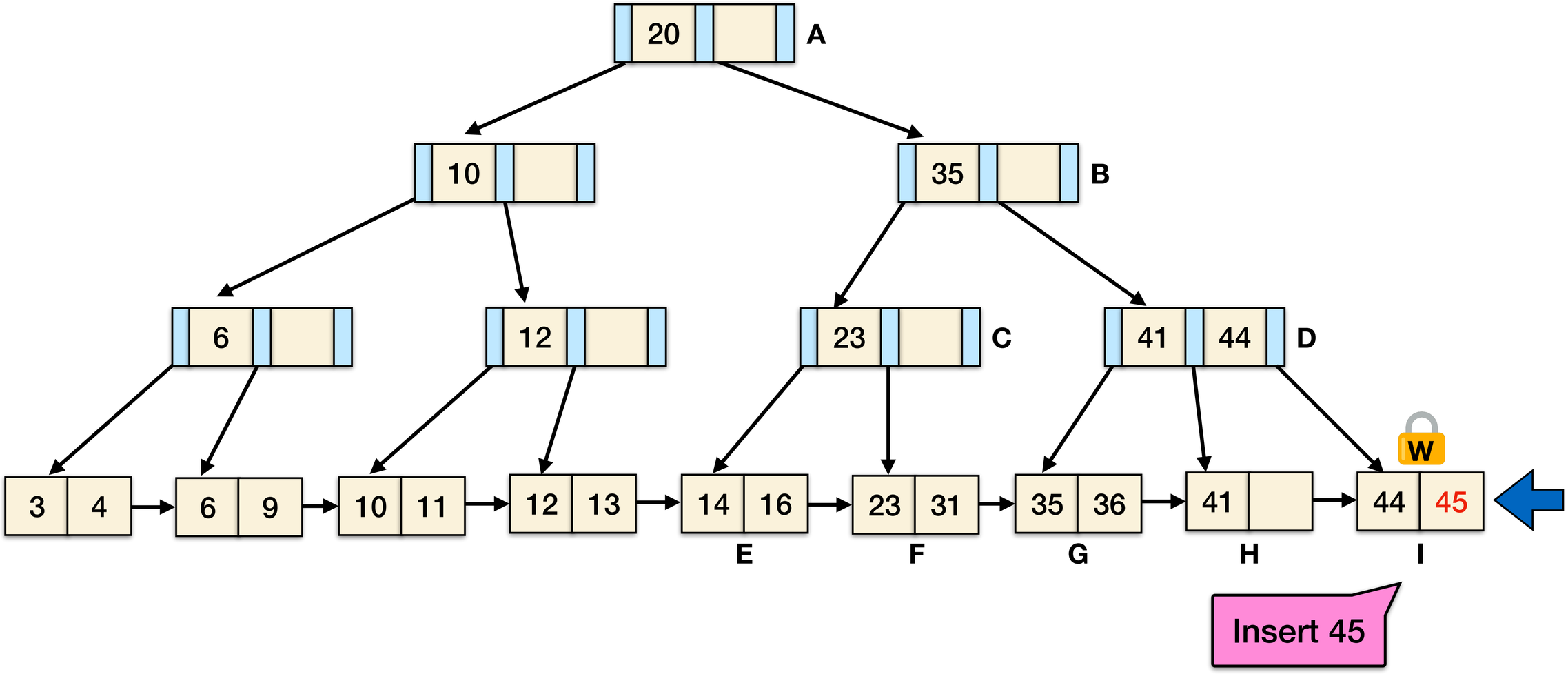
Example #3 - Insert 45



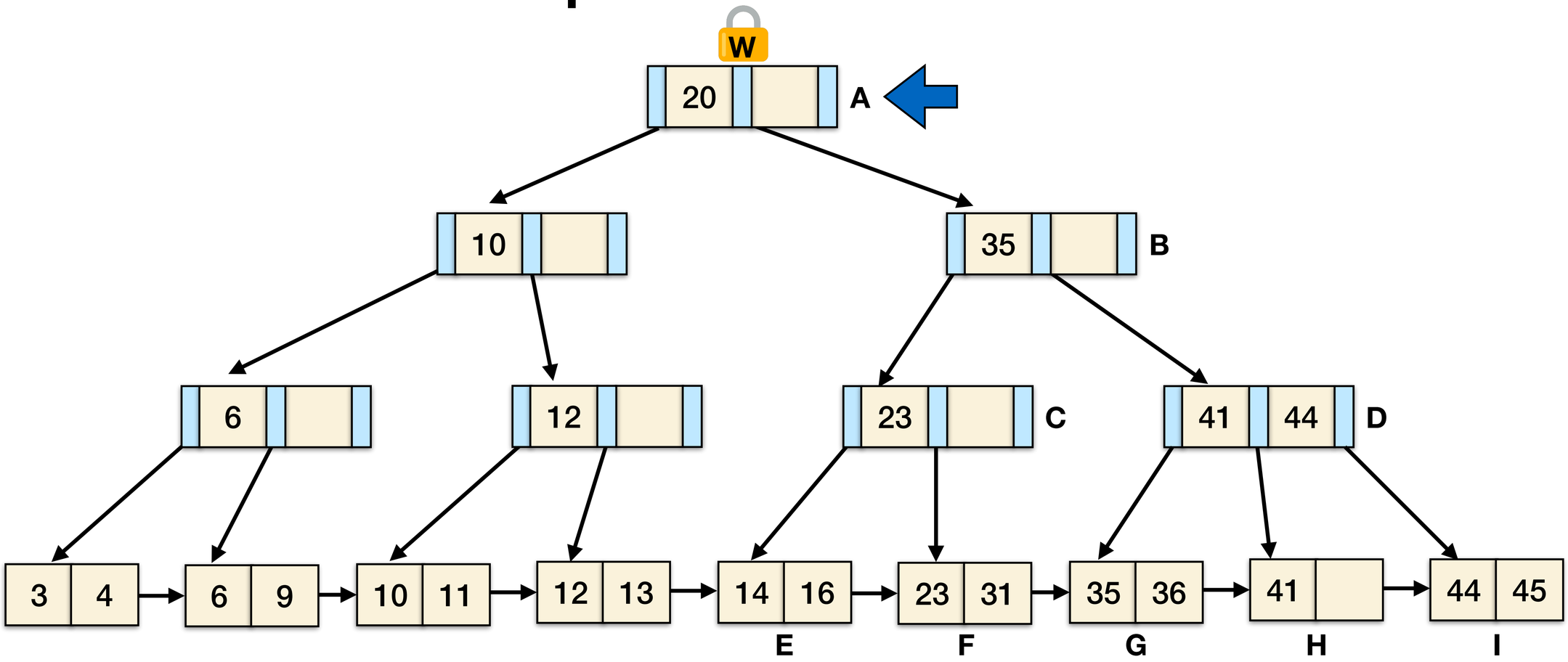
Example #3 - Insert 45



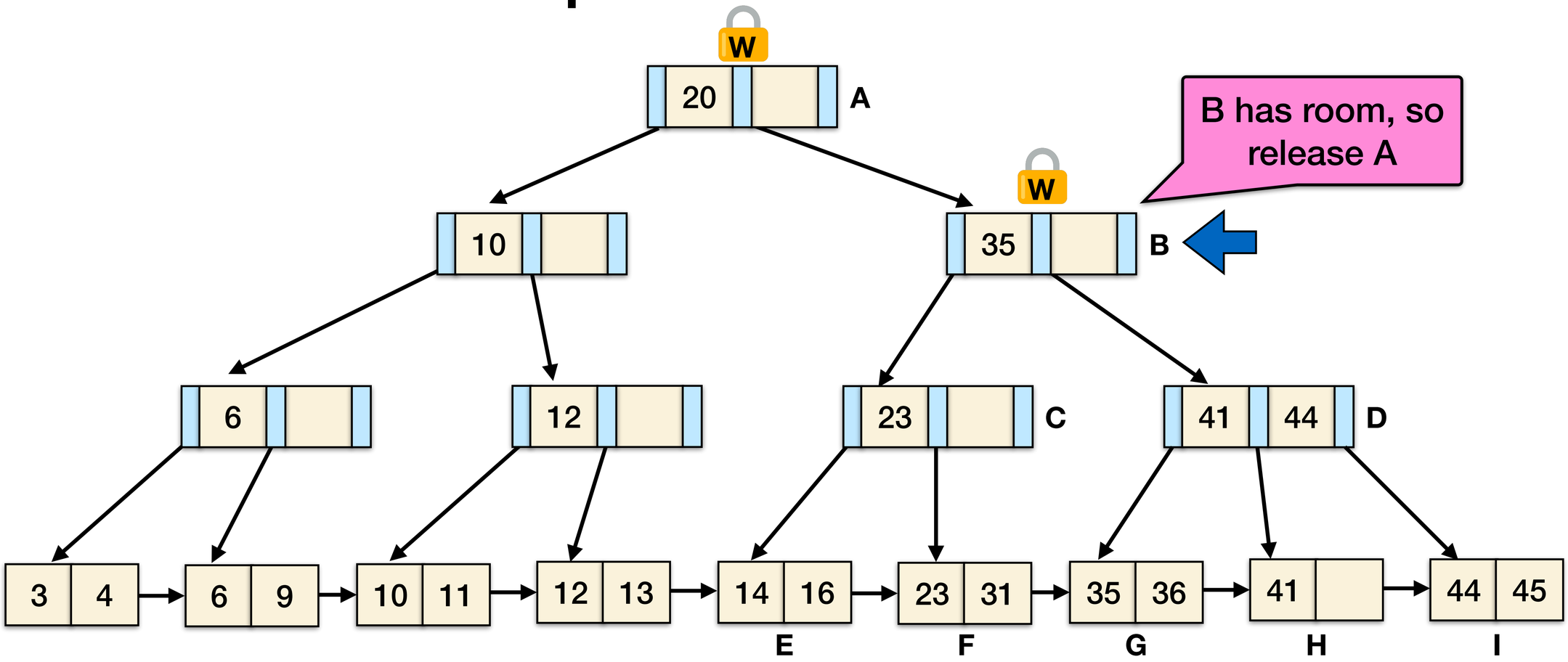
Example #3 - Insert 45



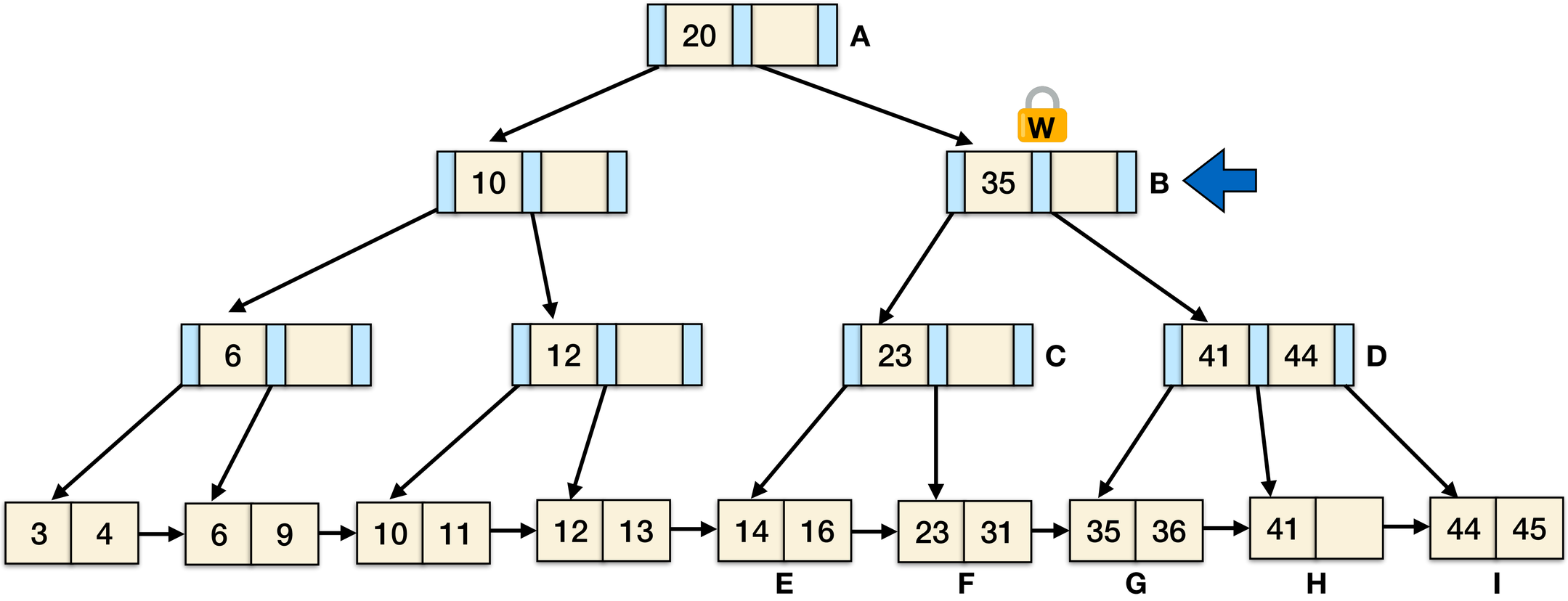
Example #4 - Insert 25



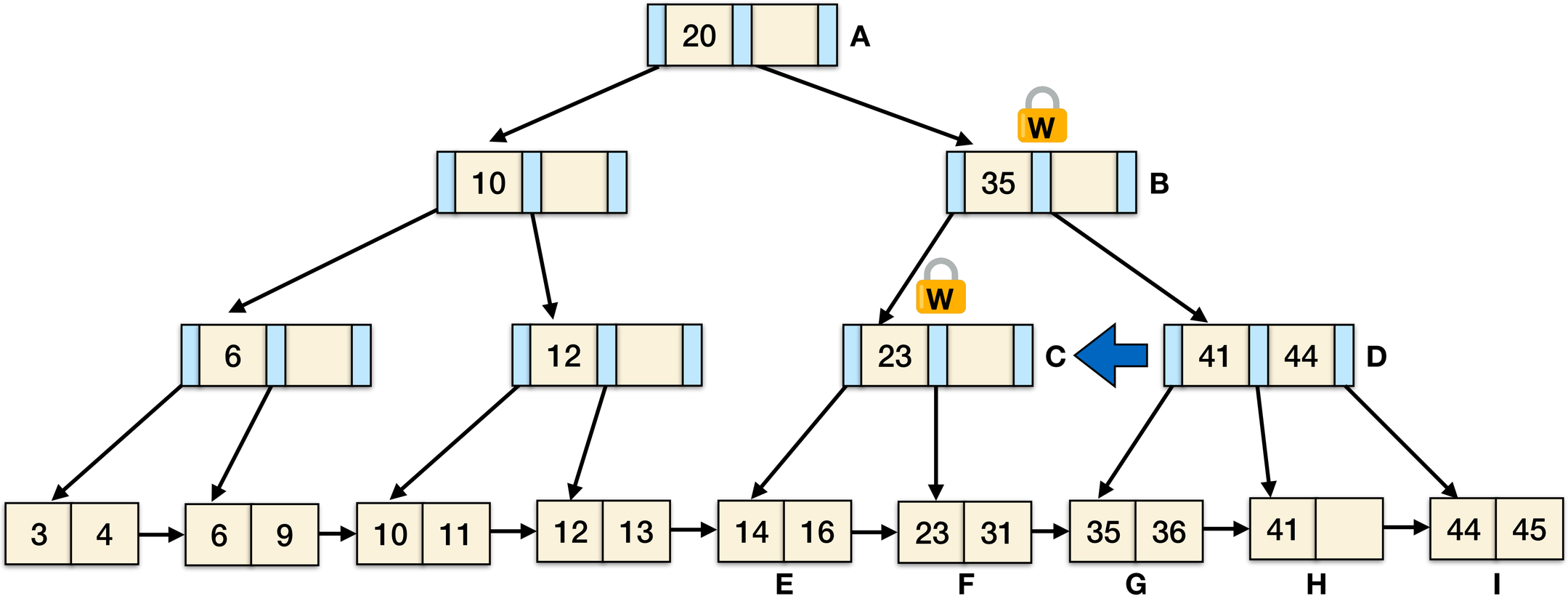
Example #4 - Insert 25



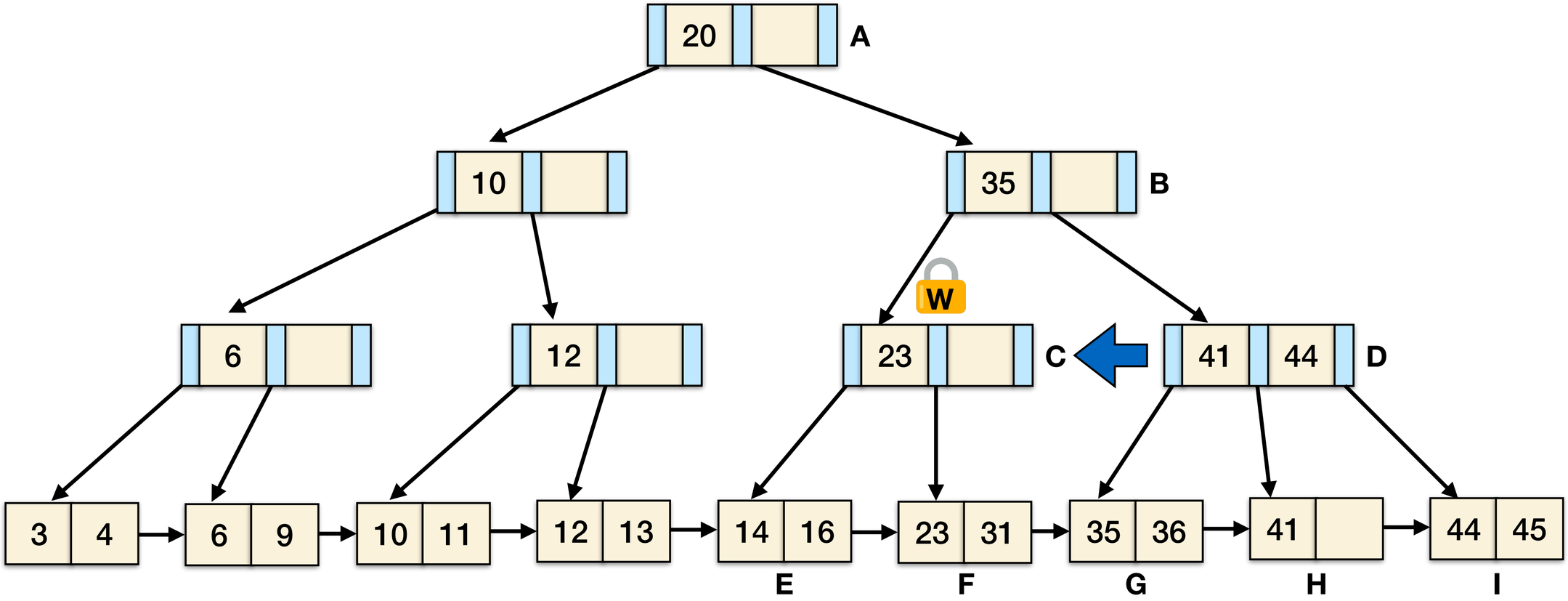
Example #4 - Insert 25



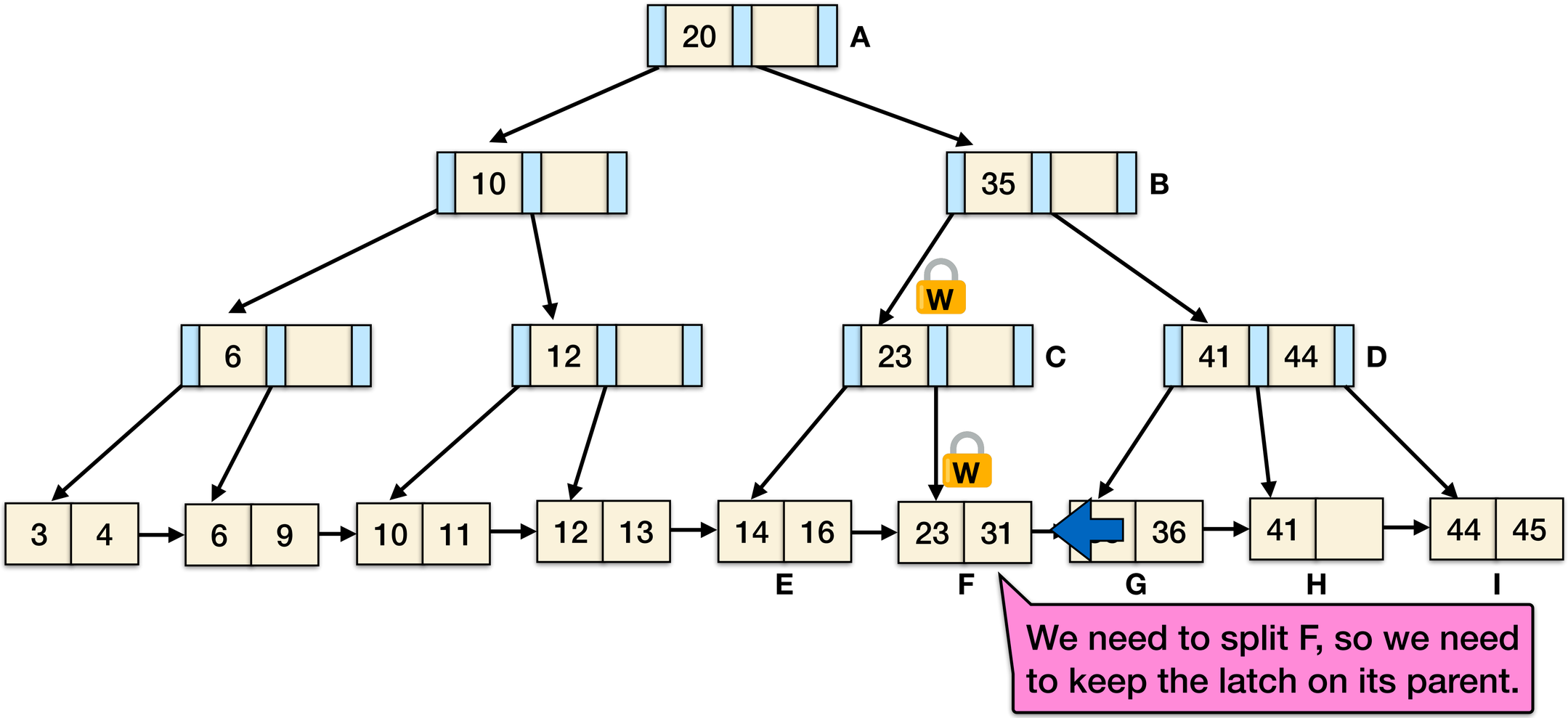
Example #4 - Insert 25



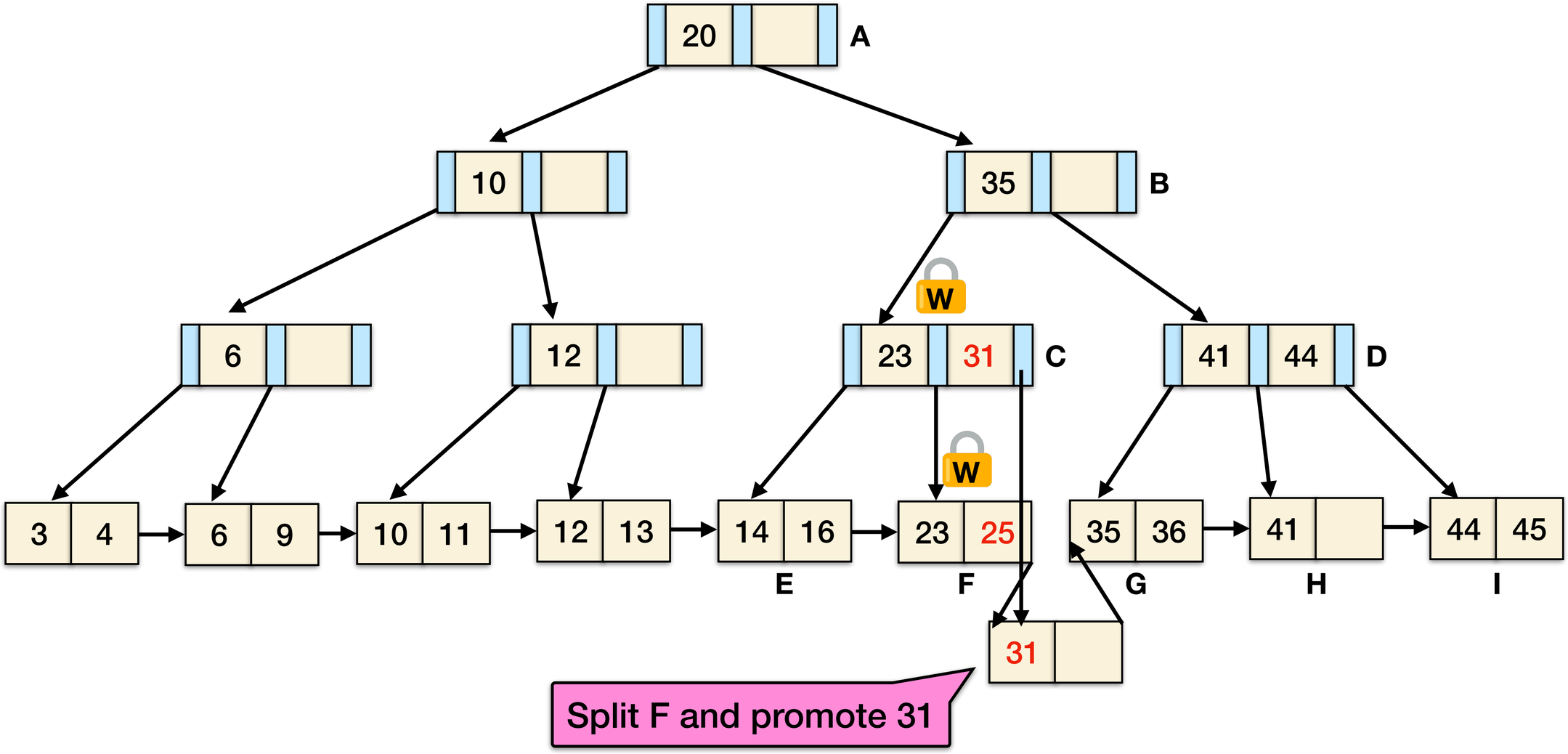
Example #4 - Insert 25



Example #4 - Insert 25



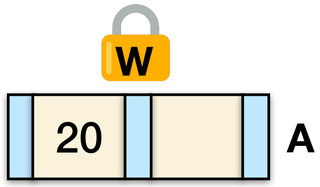
Example #4 - Insert 25



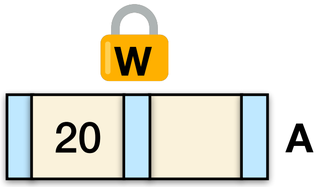
Observation

What was the first step that all the update examples did on the B+-tree?

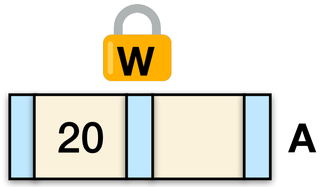
Delete 38



Insert 45



Insert 25



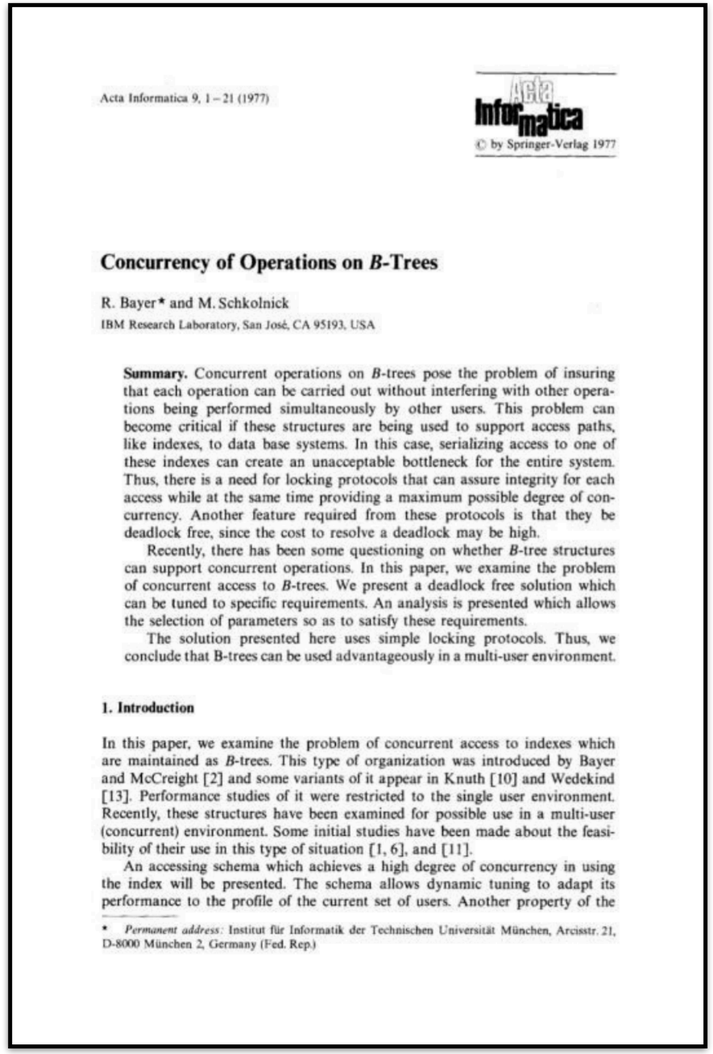
Taking a write latch on the root every time becomes a **bottleneck with higher concurrency**.

Optimistic concurrency control

Most modifications to a B+-tree will **not** require a split or merge.

Instead of assuming that there will be a split/merge, **optimistically traverse** the tree using read latches.

If you guess wrong, **repeat traversal** with the pessimistic algorithm.



Acta Informatica 9, 1–21 (1977)

Acta Informatica

Concurrency of Operations on *B*-Trees

R. Bayer* and M. Schkolnick

IBM Research Laboratory, San José, CA 95193, USA

Summary. Concurrent operations on *B*-trees pose the problem of insuring that each operation can be carried out without interfering with other operations being performed simultaneously by other users. This problem can become critical if these structures are being used to support access paths, like indexes, to data base systems. In this case, serializing access to one of these indexes can create an unacceptable bottleneck for the entire system. Thus, there is a need for locking protocols that can assure integrity for each access while at the same time providing a maximum possible degree of concurrency. Another feature required from these protocols is that they be deadlock free, since the cost to resolve a deadlock may be high.

Recently, there has been some questioning on whether *B*-tree structures can support concurrent operations. In this paper, we examine the problem of concurrent access to *B*-trees. We present a deadlock free solution which can be tuned to specific requirements. An analysis is presented which allows the selection of parameters so as to satisfy these requirements.

The solution presented here uses simple locking protocols. Thus, we conclude that B-trees can be used advantageously in a multi-user environment.

1. Introduction

In this paper, we examine the problem of concurrent access to indexes which are maintained as *B*-trees. This type of organization was introduced by Bayer and McCreight [2] and some variants of it appear in Knuth [10] and Wedekind [13]. Performance studies of it were restricted to the single user environment. Recently, these structures have been examined for possible use in a multi-user (concurrent) environment. Some initial studies have been made about the feasibility of their use in this type of situation [1, 6], and [11].

An accessing schema which achieves a high degree of concurrency in using the index will be presented. The schema allows dynamic tuning to adapt its performance to the profile of the current set of users. Another property of the

* *Permanent address:* Institut für Informatik der Technischen Universität München, Arcisstr. 21, D-8000 München 2, Germany (Fed. Rep.)

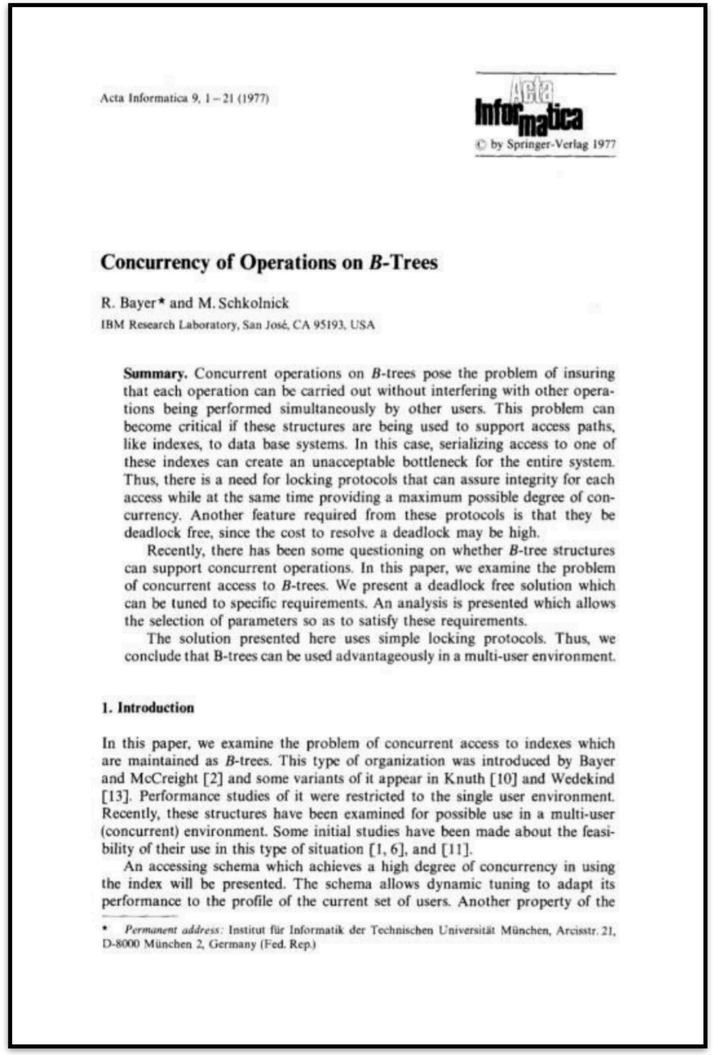
Optimistic concurrency control

Search: same as before

Insert/Delete:

- Set latches as if for search, get to leaf, and set **W** latch on leaf.
- If leaf is not safe, release all latches, and **restart thread** using previous insert/delete protocol with write latches.

This approach **optimistically assumes that only leaf node will be modified**; if not, R latches set on the first pass to leaf are wasteful



Acta Informatica 9, 1–21 (1977)

Acta Informatica

Concurrency of Operations on *B*-Trees

R. Bayer* and M. Schkolnick
IBM Research Laboratory, San José, CA 95193, USA

Summary. Concurrent operations on *B*-trees pose the problem of insuring that each operation can be carried out without interfering with other operations being performed simultaneously by other users. This problem can become critical if these structures are being used to support access paths, like indexes, to data base systems. In this case, serializing access to one of these indexes can create an unacceptable bottleneck for the entire system. Thus, there is a need for locking protocols that can assure integrity for each access while at the same time providing a maximum possible degree of concurrency. Another feature required from these protocols is that they be deadlock free, since the cost to resolve a deadlock may be high.

Recently, there has been some questioning on whether *B*-tree structures can support concurrent operations. In this paper, we examine the problem of concurrent access to *B*-trees. We present a deadlock free solution which can be tuned to specific requirements. An analysis is presented which allows the selection of parameters so as to satisfy these requirements.

The solution presented here uses simple locking protocols. Thus, we conclude that B-trees can be used advantageously in a multi-user environment.

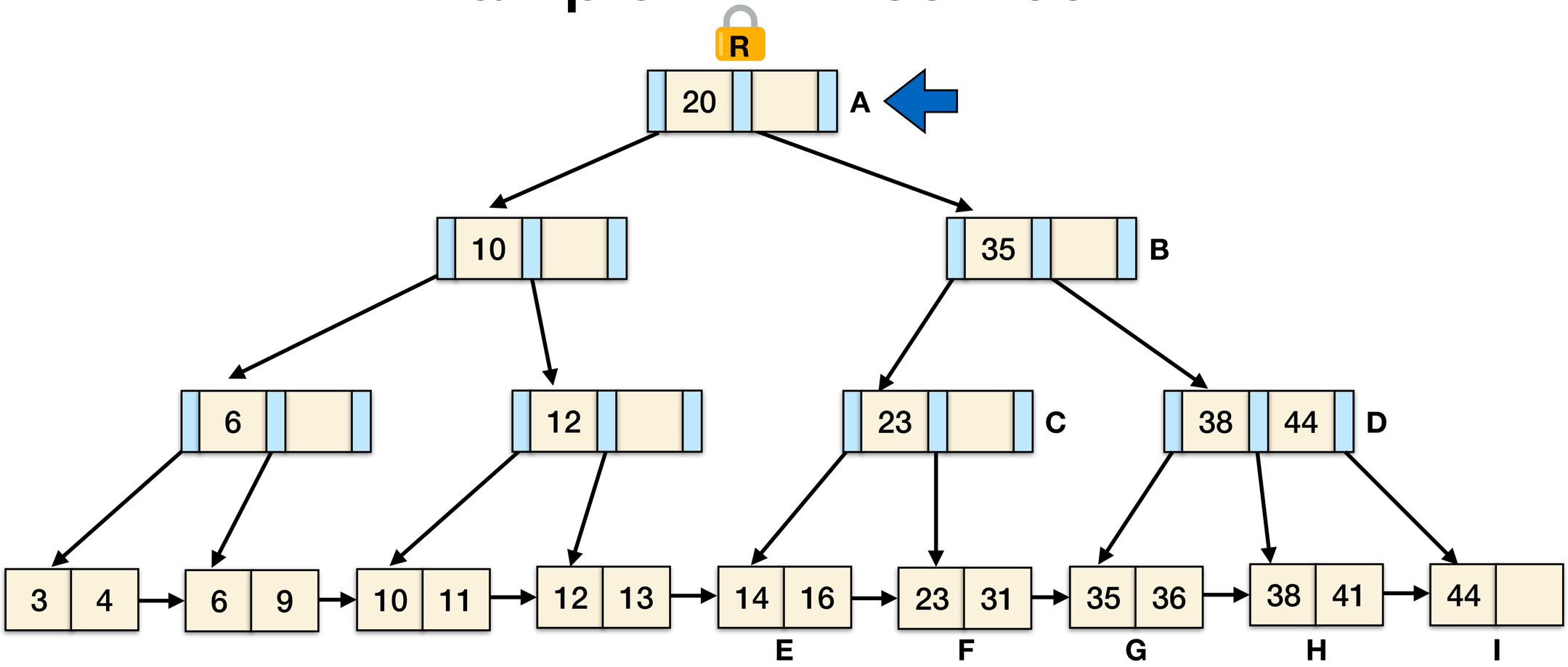
1. Introduction

In this paper, we examine the problem of concurrent access to indexes which are maintained as *B*-trees. This type of organization was introduced by Bayer and McCreight [2] and some variants of it appear in Knuth [10] and Wedekind [13]. Performance studies of it were restricted to the single user environment. Recently, these structures have been examined for possible use in a multi-user (concurrent) environment. Some initial studies have been made about the feasibility of their use in this type of situation [1, 6], and [11].

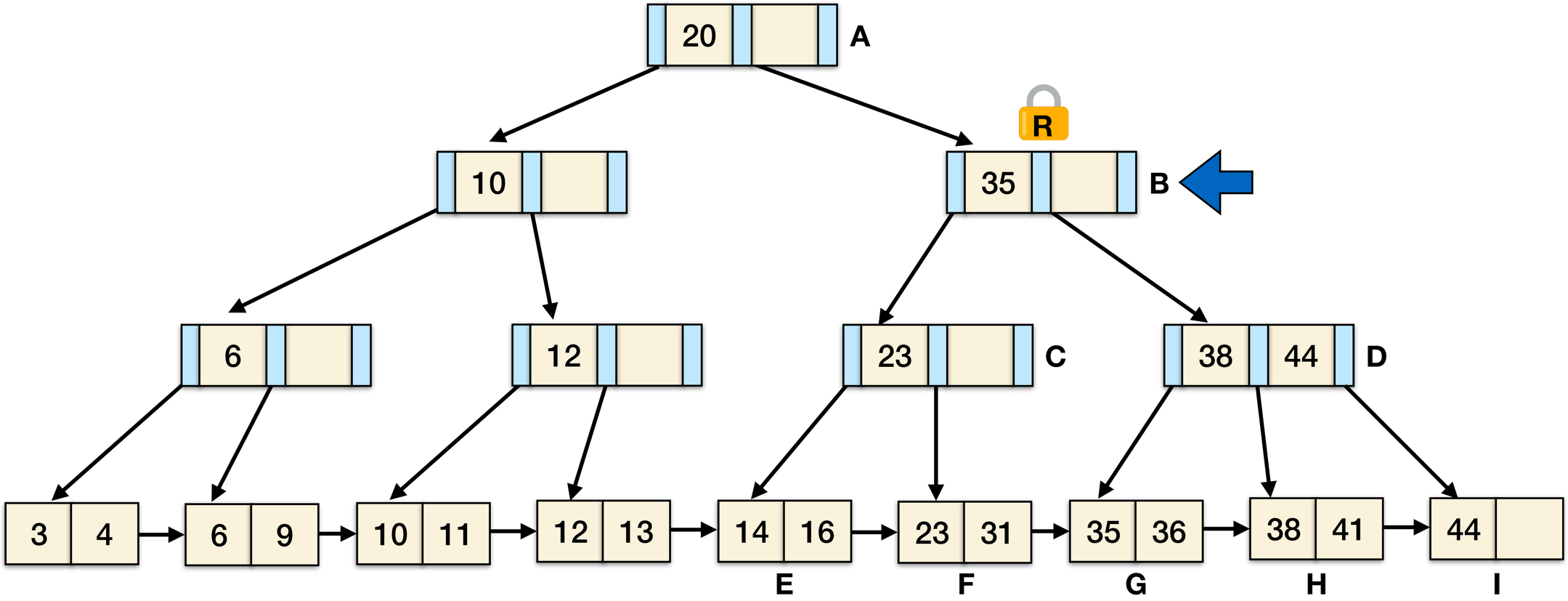
An accessing schema which achieves a high degree of concurrency in using the index will be presented. The schema allows dynamic tuning to adapt its performance to the profile of the current set of users. Another property of the

* *Permanent address:* Institut für Informatik der Technischen Universität München, Arcisstr. 21, D-8000 München 2, Germany (Fed. Rep.)

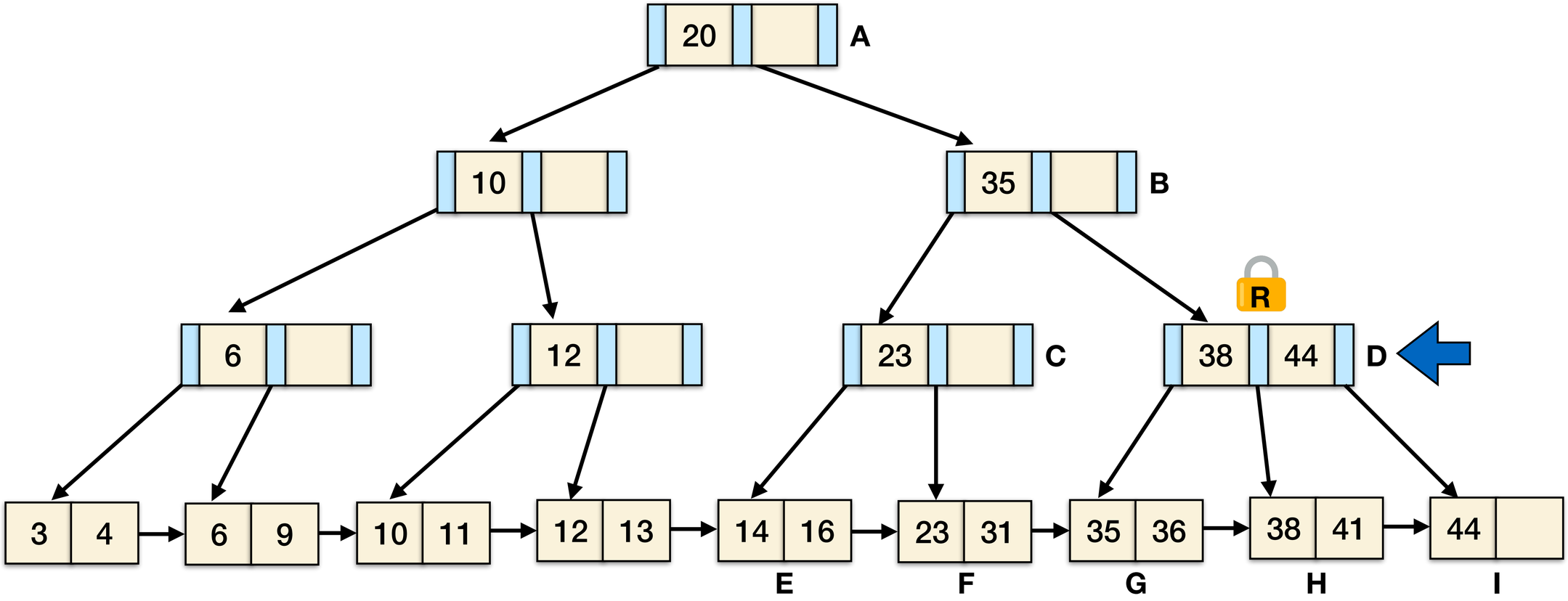
Example #2 - Insert 50



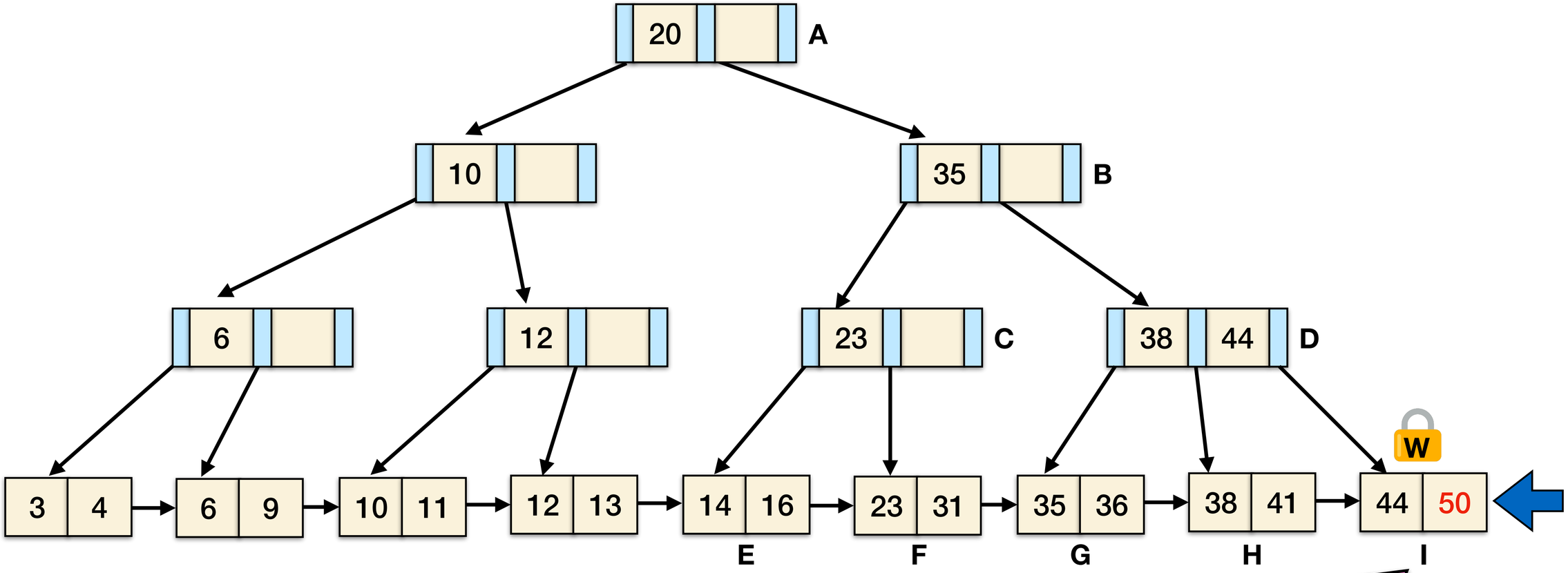
Example #2 - Insert 50



Example #2 - Insert 50

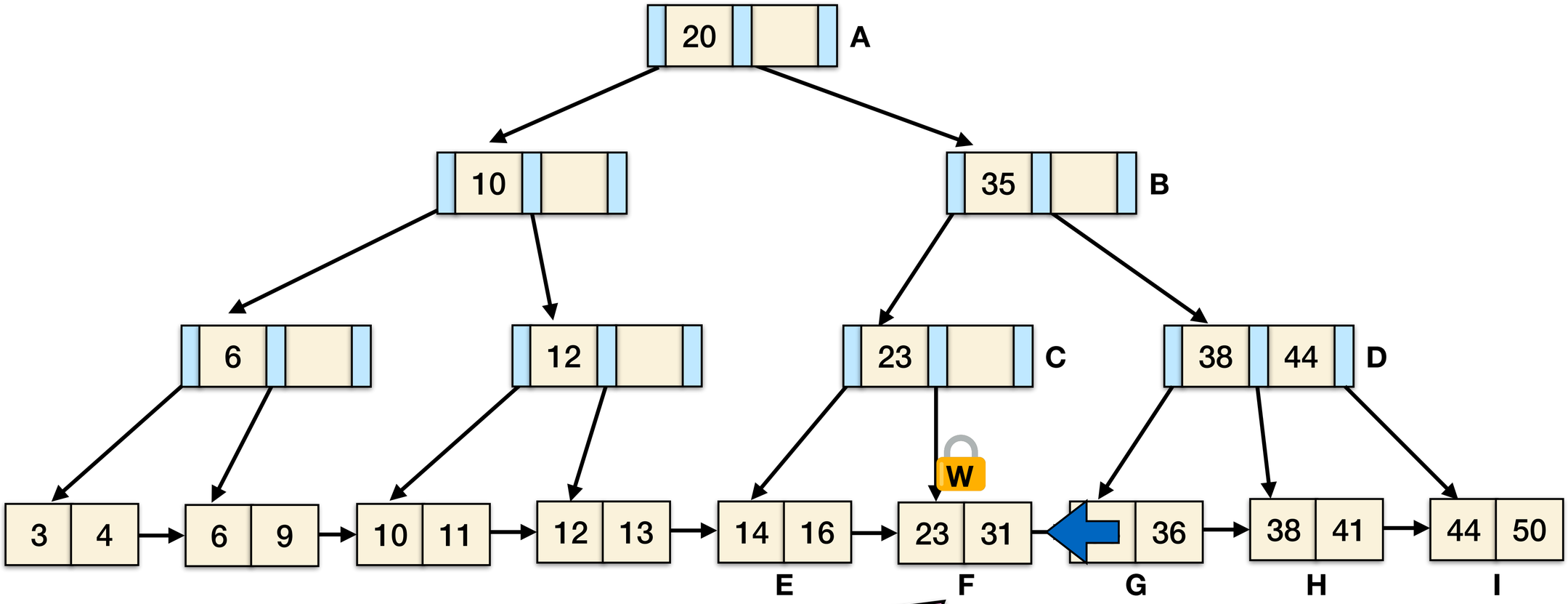


Example #2 - Insert 50



Always take the write lock on the leaves. Since this case does not split, we only need to write to the leaf, and we are done.

Example #2 - Insert 25



We need to split F, so we would have to restart from the top and take W locks all the way down.

Summary

- Locking protocols can be subtle and require thinking carefully about correctness.
- Be careful to **avoid deadlock** by respecting a total ordering of locks.
- Straightforward locking protocols are easy to reason about, but may not be good enough for high-concurrency situations.
- **Optimistic concurrency control** improves performance when there is a high level of contention.

CSE 6230:
HPC Tools and Applications

+

Lecture 8: Analysis of Multithreaded Algorithms

Helen Xu

hxu615@gatech.edu

Georgia Tech College of Computing
School of Computational Science and Engineering

What is Parallelism?

Execution Model

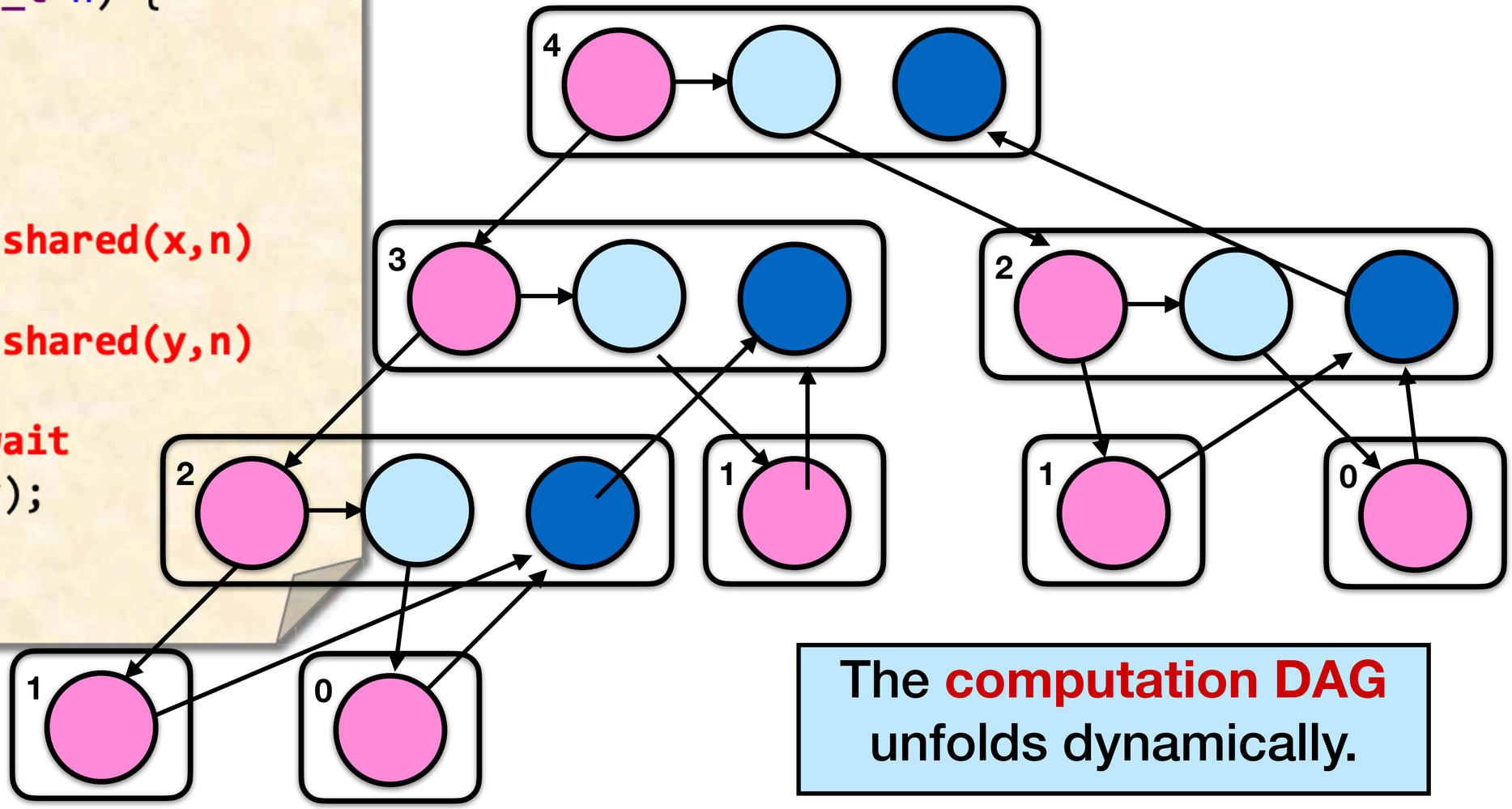
Example: fib(4)

```
int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x, y;
#pragma omp task shared(x,n)
    x = fib(n-1);
#pragma omp task shared(y,n)
    y = fib(n-2);
#pragma omp taskwait
    return (x + y);
  }
}
```

Execution Model

```
int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x, y;
#pragma omp task shared(x,n)
    x = fib(n-1);
#pragma omp task shared(y,n)
    y = fib(n-2);
#pragma omp taskwait
    return (x + y);
  }
}
```

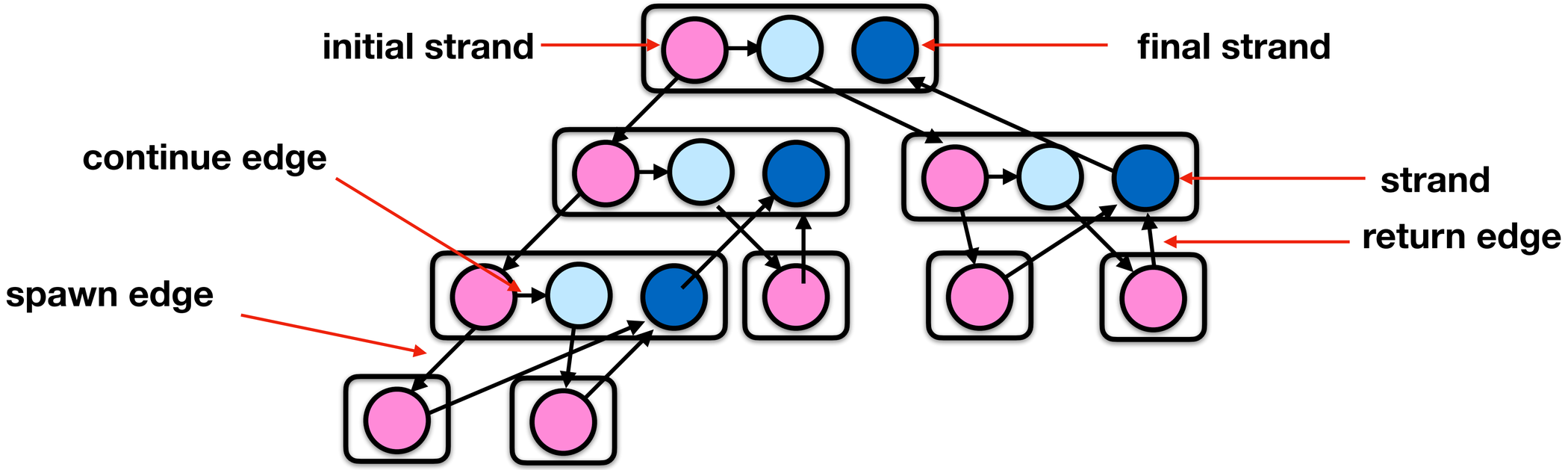
Example: fib(4)



"Processor oblivious"

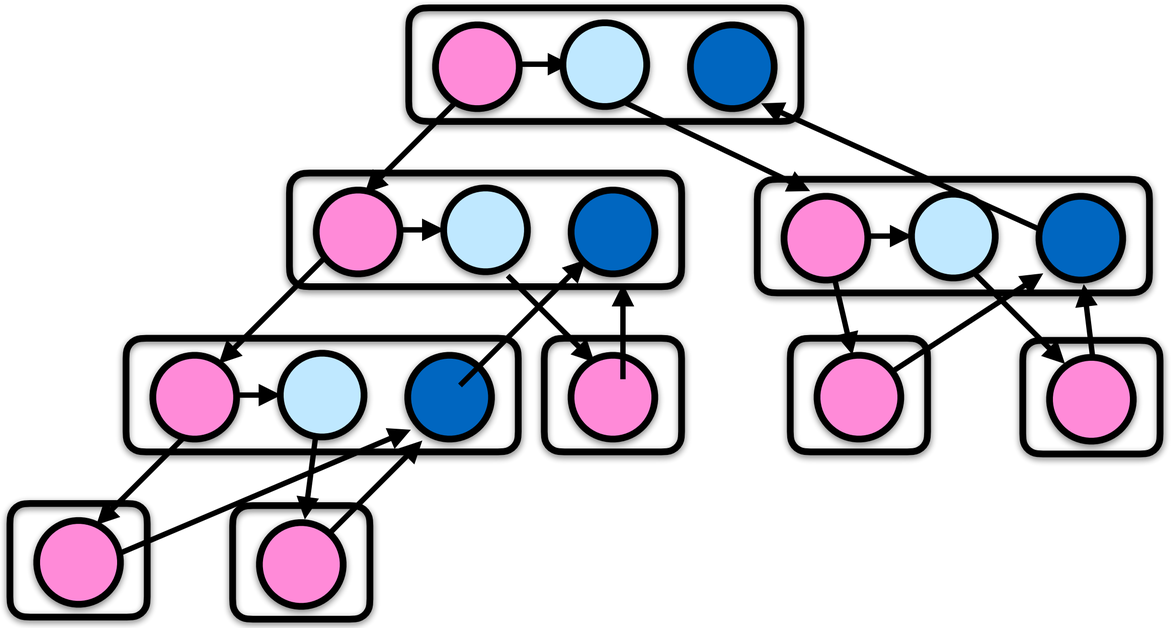
The **computation DAG** unfolds dynamically.

Computation DAG



- A parallel instruction stream is a dag G = (V, E).
- Each vertex v ∈ V is a **strand**: a sequence of instructions not containing a spawn, sync, or return from a spawn.
- An edge e ∈ E is a spawn, call (function), return, or continue edge.
- Loop parallelism (`parallel for`) can be converted (for analysis purposes) to task spawn and wait using recursive divide-and-conquer.

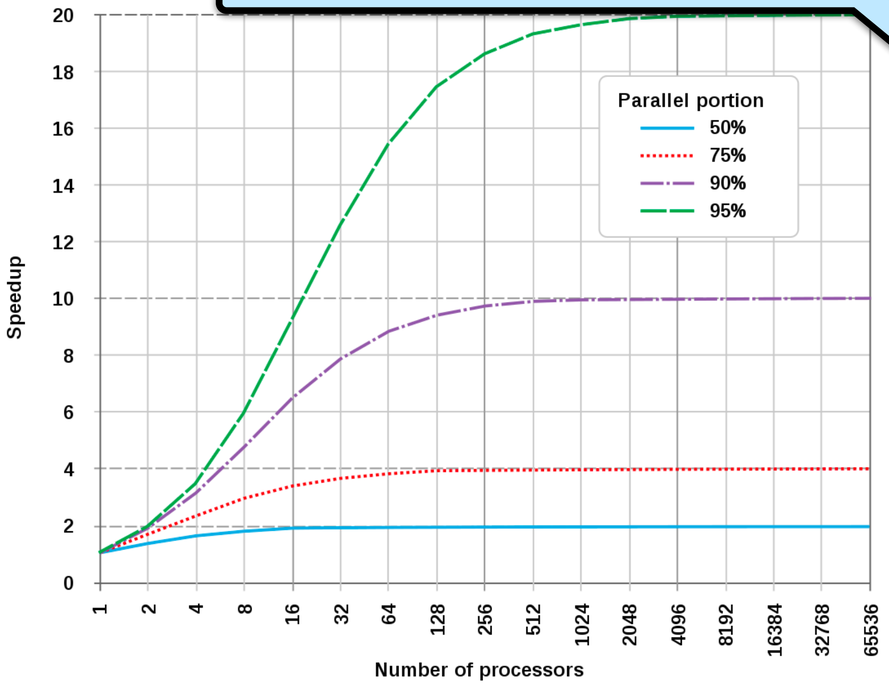
How Much Parallelism?



Assuming that each strand executes in unit time, what is the **parallelism** of this computation?

Recall: Amdahl's Law

Work done by the best sequential algorithm



Amdahl's law bounds **strong scaling**, or the speedup of a fixed problem given more parallel resources.

s = fraction of work done seqentially (Amdahl fraction)
1-s = parallelizable fraction
P = number of processors

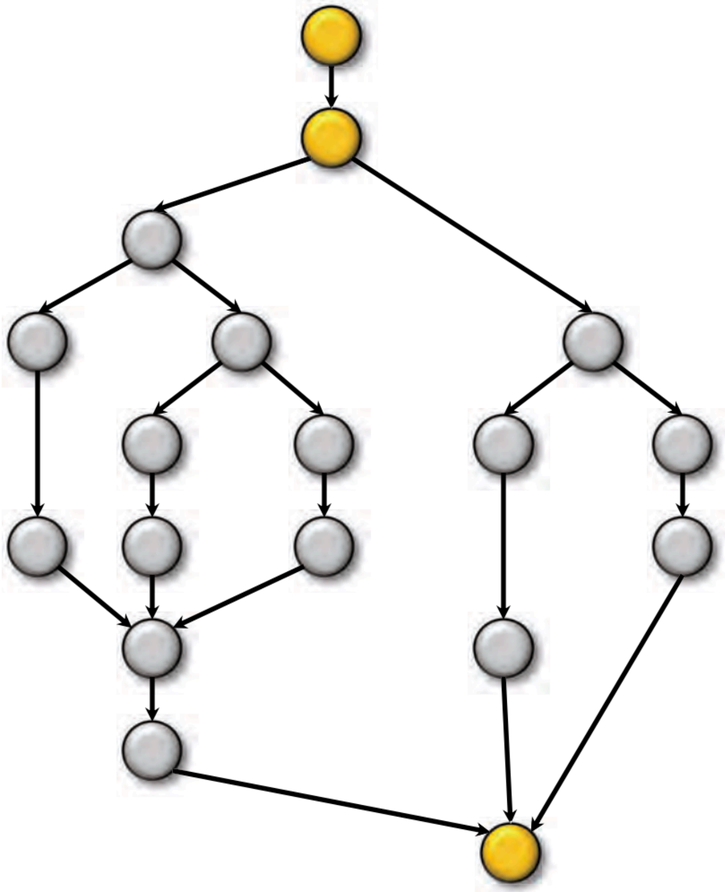
Speedup (P) = Time(1) / Time(P)
<= 1/(s + (1-s)/P)
<= 1/s

For a fixed problem, the upper limit of speedup is **determined by the serial fraction.**

From Ramki Kannan

Quantifying Parallelism

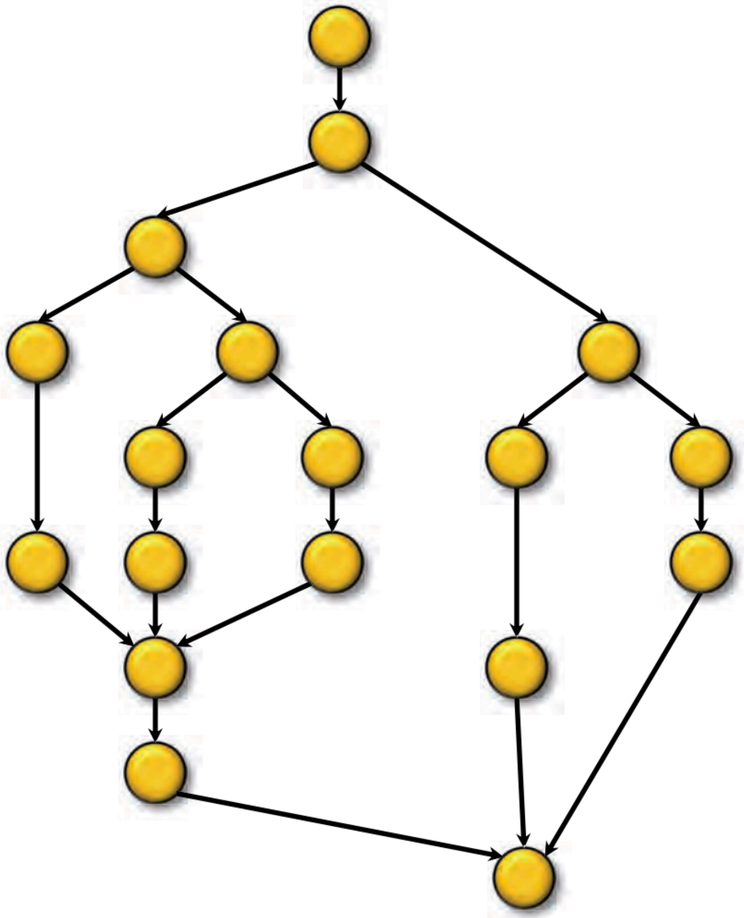
What is the **parallelism** of this computation?



Amdahl's Law says that since the serial fraction is 3/18 = 1/6, the speedup is upper bounded by 6.

Performance Measures

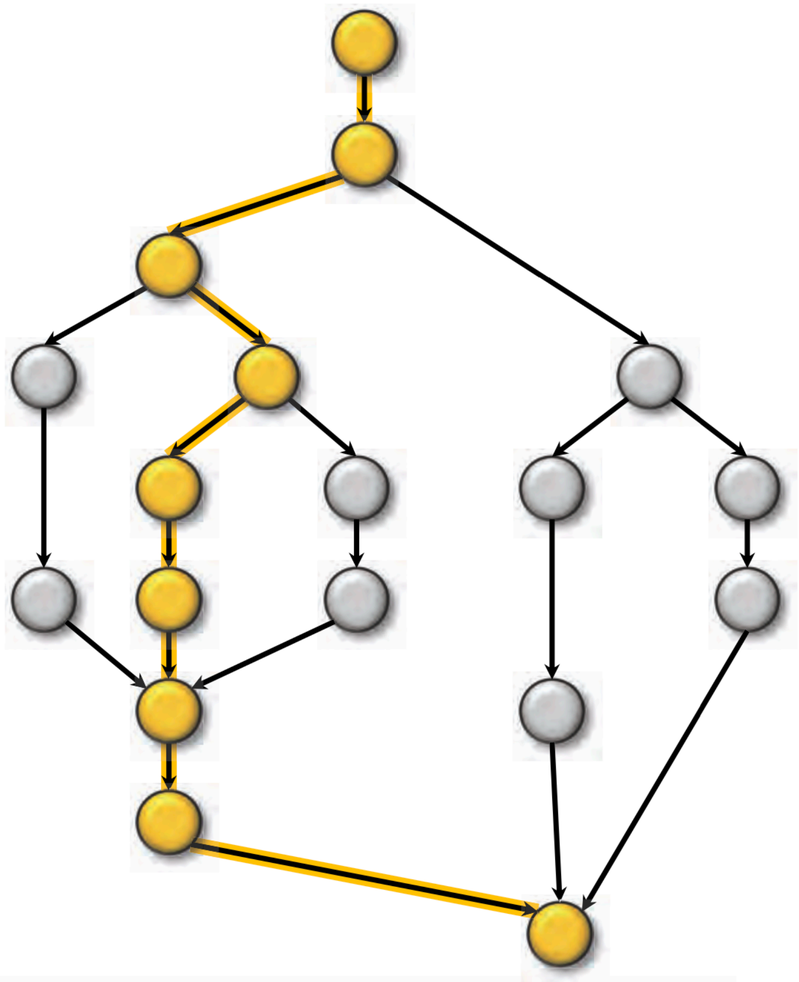
T_p = execution time on P processors



T_1 = **work**
= 18

Performance Measures

T_p = execution time on P processors



T_1 = **work**
= 18

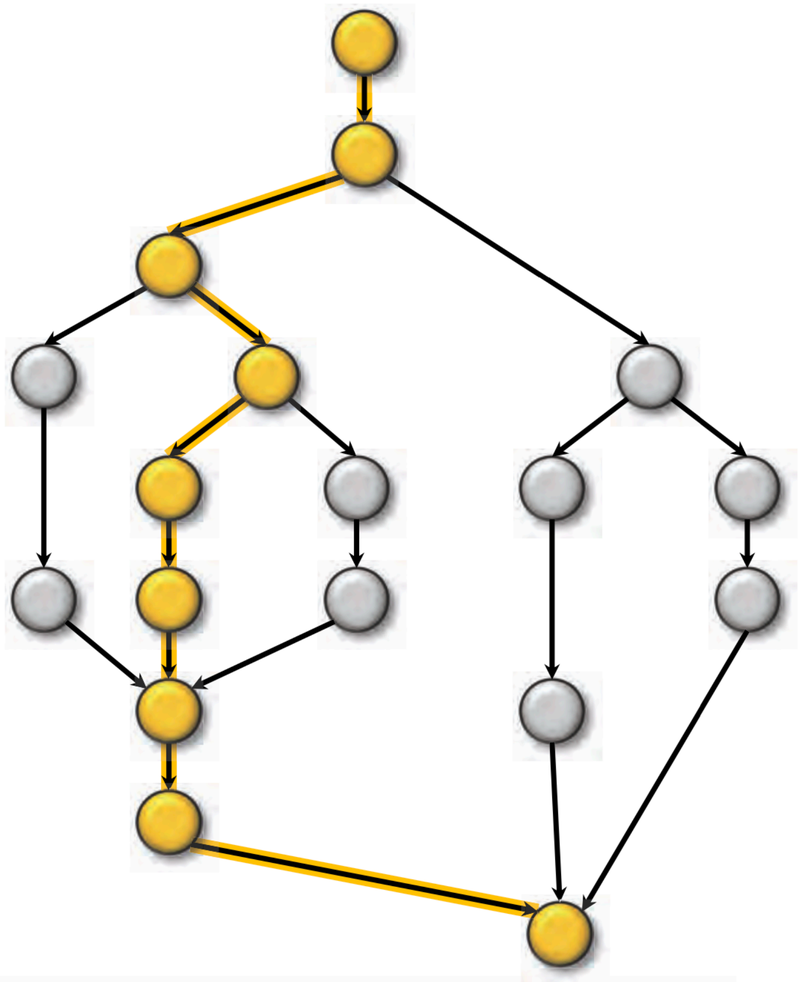
T_∞ = **span***
= 9

Time with an infinite number of processors

* Also called **critical-path length** or **computational depth**

Performance Measures

T_p = execution time on P processors



Time with an infinite number of processors

T_1 = **work**
= 18

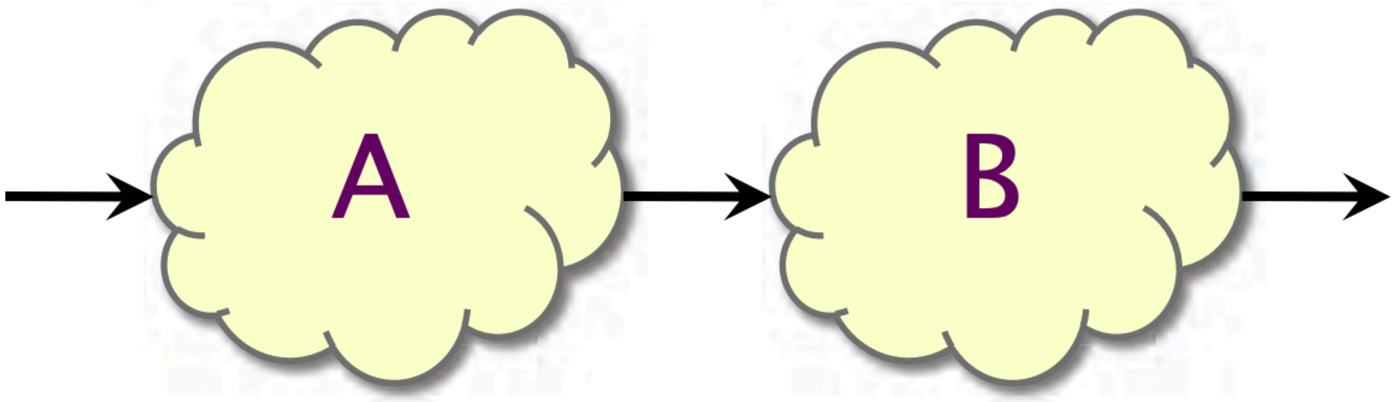
T_∞ = **span***
= 9

Work Law
$T_p \geq T_1/P$

Span Law
$T_p \geq T_\infty$

* Also called **critical-path length** or **computational depth**

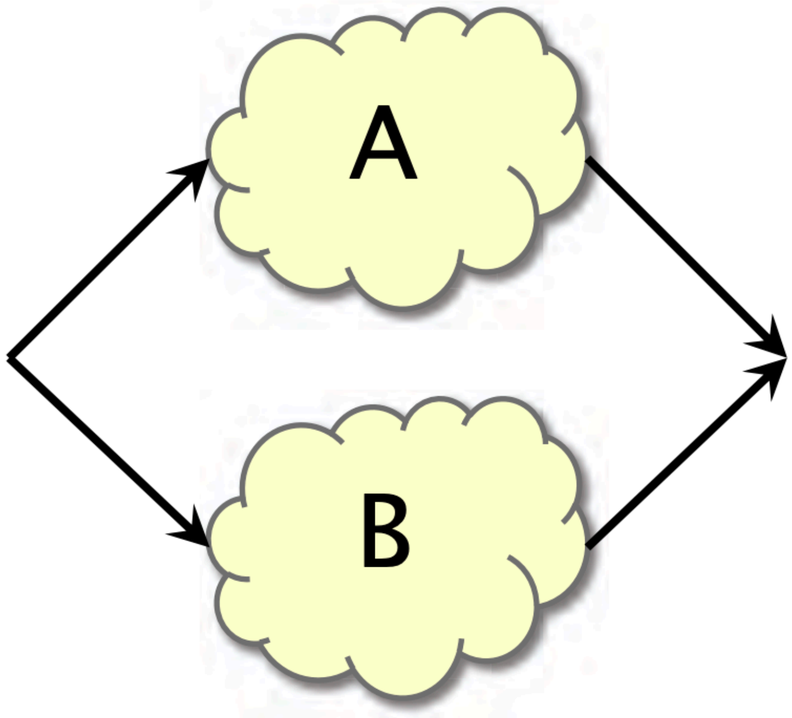
Series Composition



Work: $T_1(A \cup B) = T_1(A) + T_1(B)$

Span: $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

Parallel Composition



Work: $T_1(A \cup B) = T_1(A) + T_1(B)$

Span: $T_\infty(A \cup B) = \max\{T_\infty(A), T_\infty(B)\}$

Speedup

Definition. T_1/T_P = speedup on P processors.

- If $T_1/T_P < P$, we have sublinear speedup.
- If $T_1/T_P = P$, we have (perfect) linear speedup.
- If $T_1/T_P > P$, we have superlinear speedup, which is not possible in this simple performance model, because of the WORK LAW $T_P \geq T_1/P$.

Brent's Law

Analysis of parallel algorithms usually assumes that there is an **unbounded number of processors**.

Although the assumption is unrealistic, it is not a problem in a theoretical since, since any computation that can run in parallel on N processors can be **simulated on p<N** processors by letting each processor execute multiple units of work.

Brent's Law

$$\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty$$

Parallelism

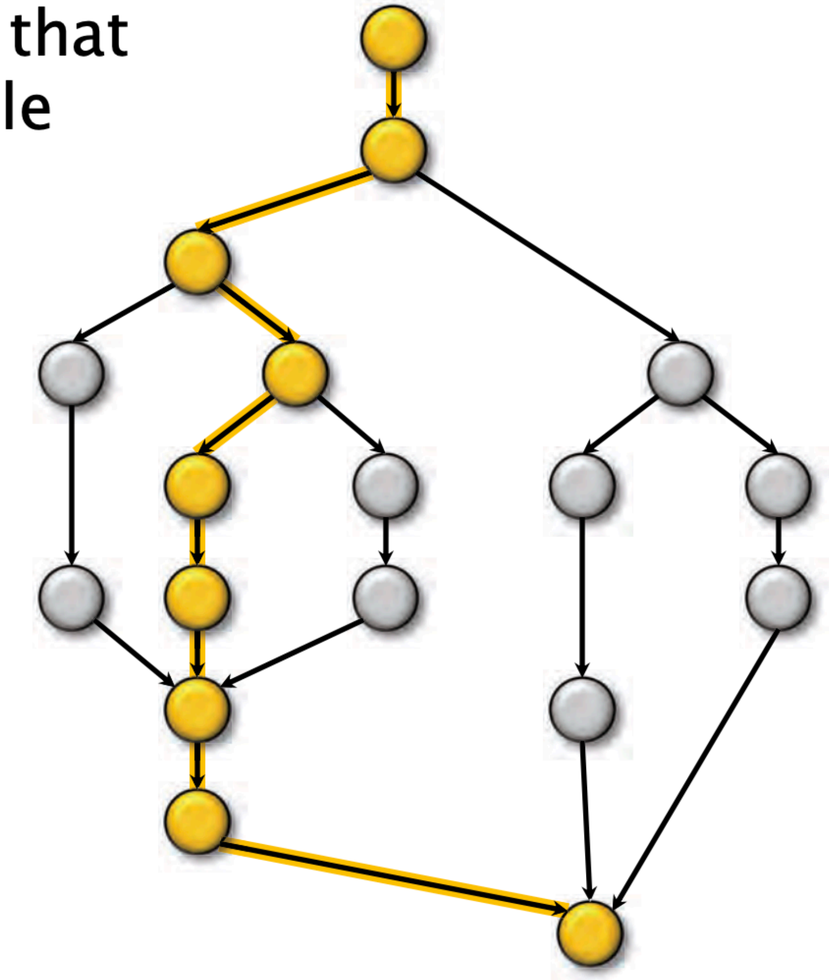
Because the SPAN LAW dictates that $T_P \geq T_\infty$, the maximum possible speedup given T_1 and T_∞ is

T_1/T_∞ = parallelism

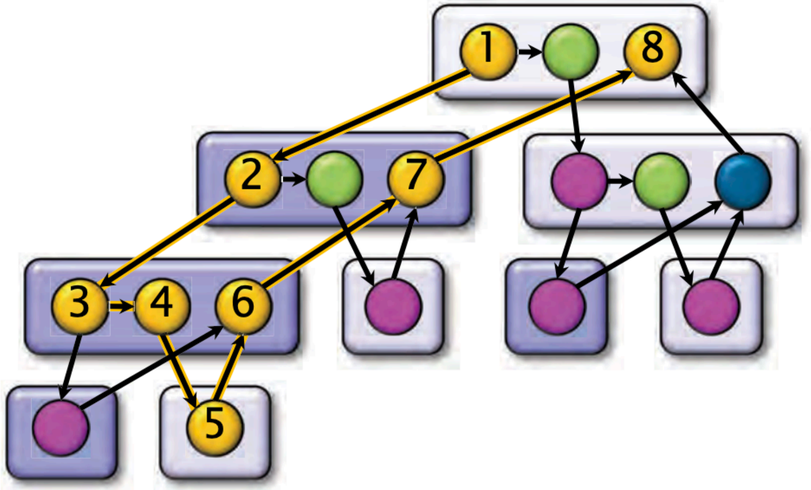
= the average amount of work per step along the span

= 18/9

= 2 .



Example: fib(4)



Assume for simplicity that each strand in `fib(4)` takes unit time to execute.

Work: $T_1 = 17$

Span: $T_\infty = 8$

Parallelism: $T_1/T_\infty = 2.125$

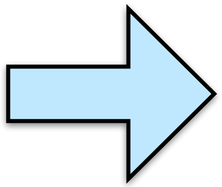
Using many more than 2 processors can yield only marginal performance gains.

Example: Parallel Quicksort

```
static void quicksort(size_t *left, size_t* right) {
  if (left == right) return;
  size_t *p = partition(left, right); // run serially
#pragma omp task
  quicksort(left, p);
#pragma omp task
  quicksort(p+1, right);
#pragma omp taskwait
}
```

Expected work = $\Theta(n \lg n)$

Expected span = $\Theta(n)$



Parallelism = $\Theta(\lg n)$

From MIT 6.172

Interesting Practical Algorithms

Algorithm	Work	Span	Parallelism
Merge sort	$\Theta(n \lg n)$	$\Theta(\lg^3 n)$	$\Theta(n/\lg^2 n)$
Matrix multiplication	$\Theta(n^3)$	$\Theta(\lg n)$	$\Theta(n^3/\lg n)$
Strassen	$\Theta(n^{\lg 7})$	$\Theta(\lg^2 n)$	$\Theta(n^{\lg 7}/\lg^2 n)$
LU-decomposition	$\Theta(n^3)$	$\Theta(n \lg n)$	$\Theta(n^2/\lg n)$
Tableau construction	$\Theta(n^2)$	$\Theta(n^{\lg 3})$	$\Theta(n^{2-\lg 3})$
FFT	$\Theta(n \lg n)$	$\Theta(\lg^2 n)$	$\Theta(n/\lg n)$
Breadth-first search	$\Theta(E)$	$\Theta(\Delta \lg V)$	$\Theta(E/\Delta \lg V)$

From MIT 6.172

Divide-and-Conquer Recurrences

The Master Method

The Master Method for solving divide-and-conquer recurrences applies to recurrences of the form*

$$T(n) = aT(n/b) + f(n) ,$$

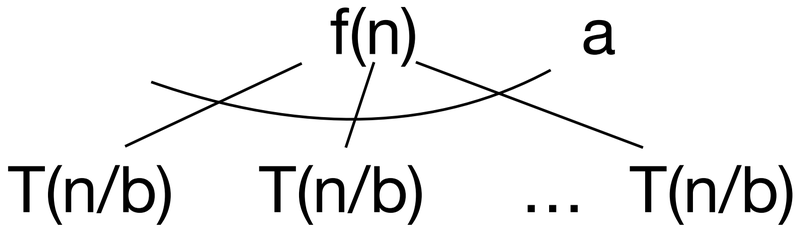
where $a \geq 1$, $b > 1$, and f is asymptotically positive.

*The unstated base case is $T(n) = \Theta(1)$ for sufficiently small n.

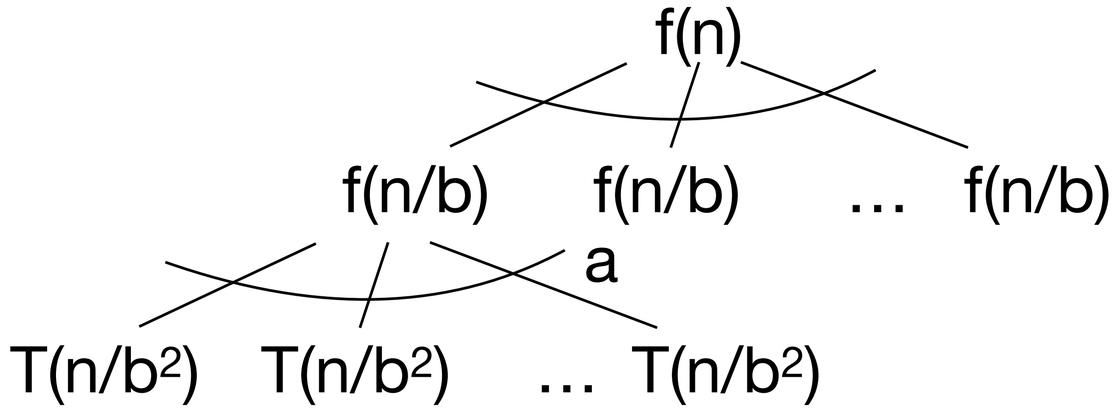
Recursion Tree: T(n) = aT(n/b) + f(n)

T(n)

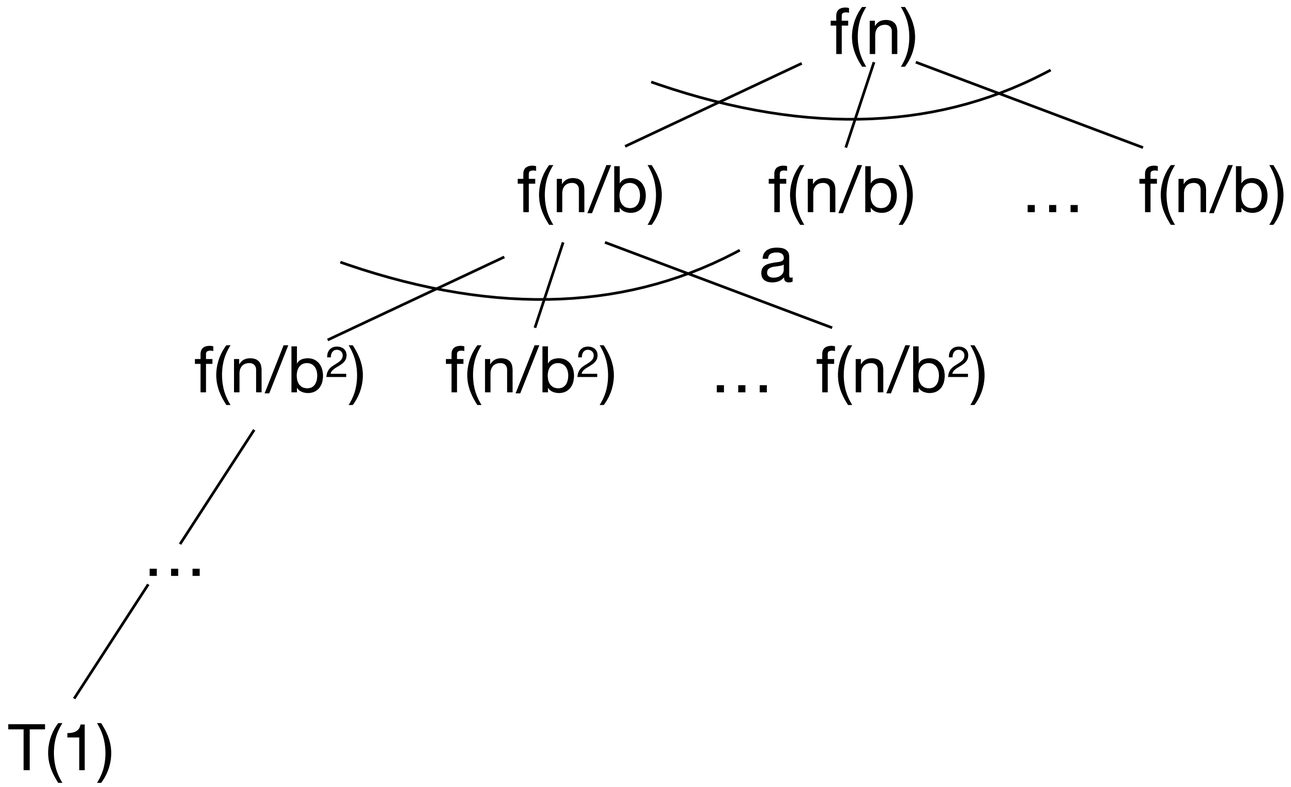
Recursion Tree: T(n) = aT(n/b) + f(n)



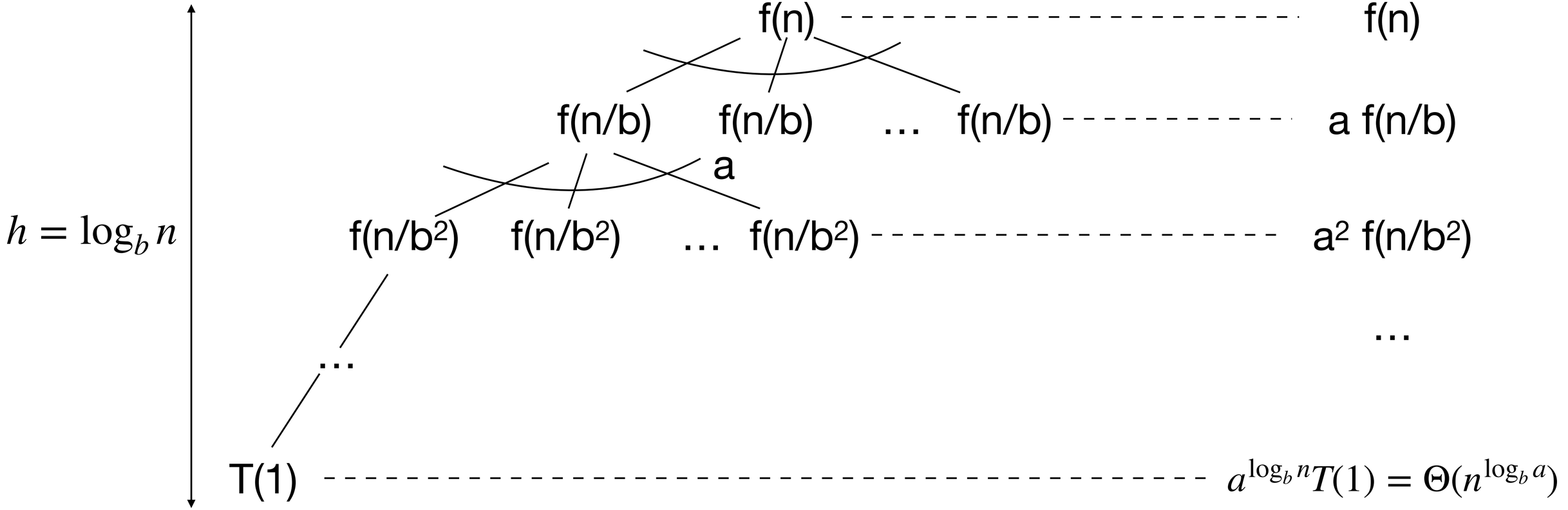
Recursion Tree: T(n) = aT(n/b) + f(n)



Recursion Tree: T(n) = aT(n/b) + f(n)

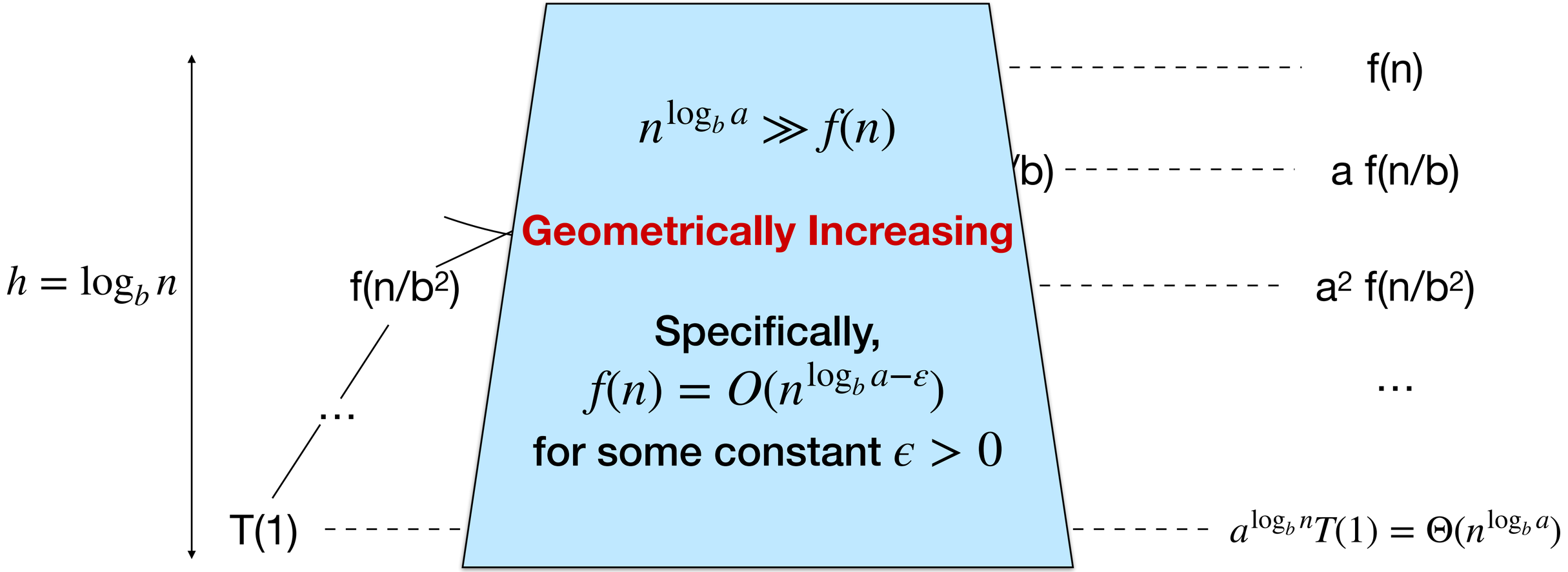


Recursion Tree: T(n) = aT(n/b) + f(n)



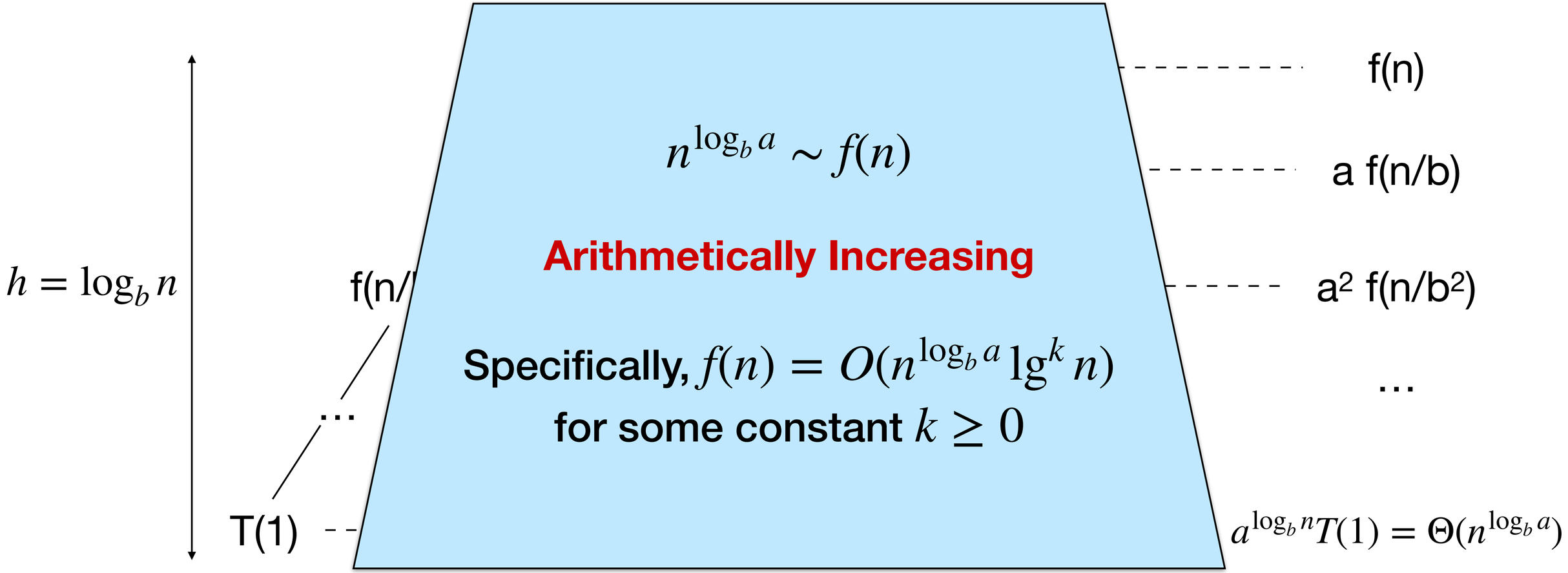
Idea: Compare $n^{\log_b a}$ with f(n).

Master Method - Case 1



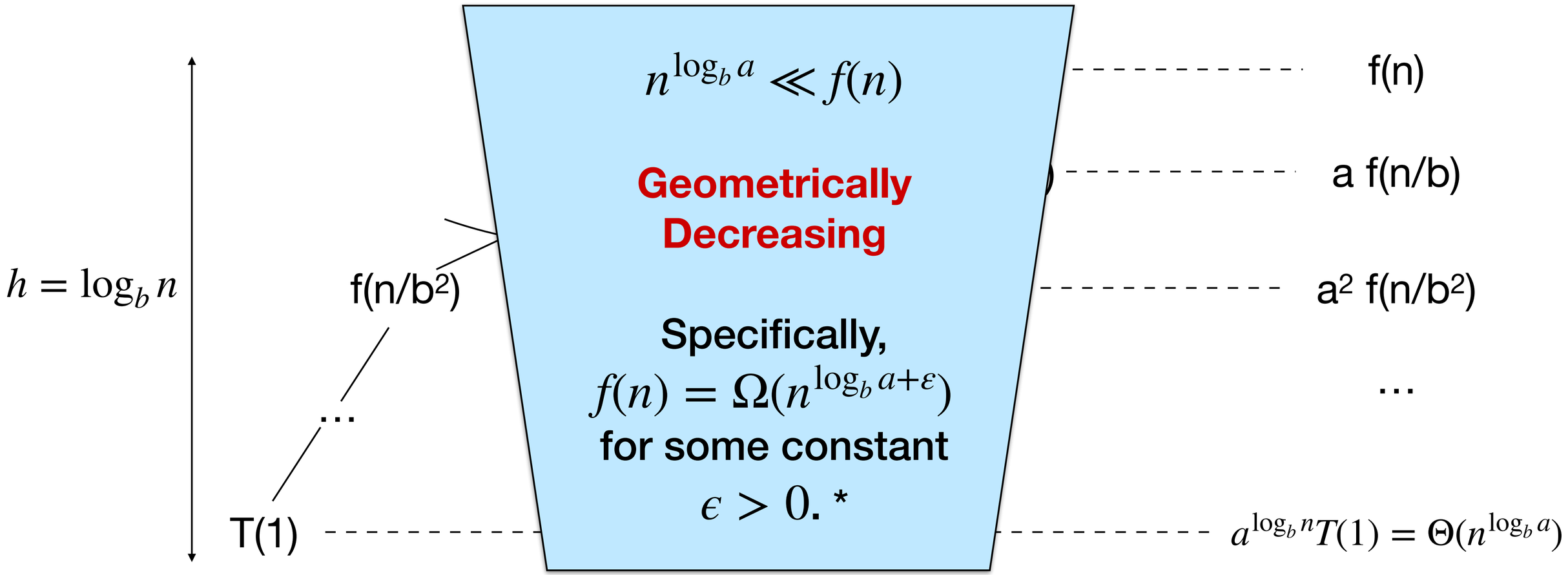
$$T(n) = \Theta(n^{\log_b a})$$

Master Method - Case 2



$$T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$$

Master Method - Case 3



$$T(n) = \Theta(f(n))$$

*and f(n) satisfies the regularity condition that a f(n/b) ≤ c f(n) for some constant c < 1.

Master-Method Cheat Sheet

Solve

$$T(n) = a\,T(n/b) + f(n)\ ,$$

where $a \geq 1$ and $b > 1$.

CASE 1: $f(n) = O(n^{\log_b a - \epsilon})$, constant $\epsilon > 0$
$\Rightarrow T(n) = \Theta(n^{\log_b a})$.

CASE 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, constant $k \geq 0$
$\Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

CASE 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$, constant $\epsilon > 0$
(and regularity condition)
$\Rightarrow T(n) = \Theta(f(n))$.

Master Method Quiz

- $T(n) = 4T(n/2) + n$
  $n^{\log_b a} = n^2 \gg n \Rightarrow$ CASE 1: $T(n) = \Theta(n^2)$.
- $T(n) = 4T(n/2) + n^2$
  $n^{\log_b a} = n^2 = n^2 \lg^0 n \Rightarrow$ CASE 2: $T(n) = \Theta(n^2 \lg n)$.
- $T(n) = 4T(n/2) + n^3$
  $n^{\log_b a} = n^2 \ll n^3 \Rightarrow$ CASE 3: $T(n) = \Theta(n^3)$.
- $T(n) = 4T(n/2) + n^2/\lg n$
  Master method does not apply!
  Answer is $T(n) = \Theta(n^2 \lg\lg n)$. (Prove by substitution.)

More general (but more complicated) solution: Akra–Bazzi method.

Parallel Loops

Analysis of Parallel Loops

Assuming each iteration takes O(1) work

A parallel for loop with n iterations has **span**:

- **O(1)** in Parallel Random-Access Machine Model (PRAM)
- **O(lg n)** in Cilk model - implemented with divide-and-conquer

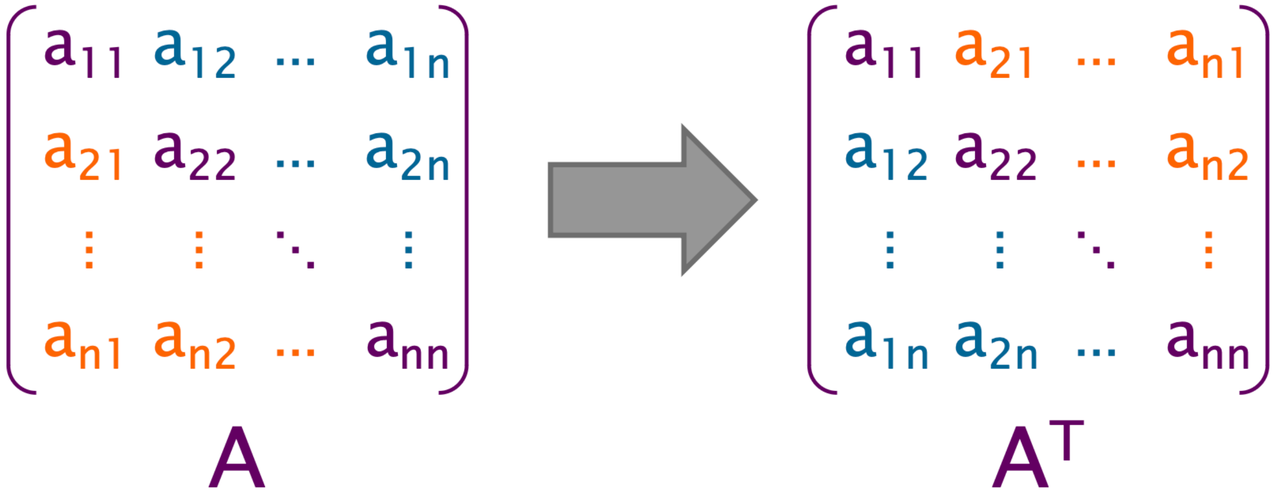
Assuming no limit on processors in the machine

Span bounds in the Cilk model are at most **polylog factors** off of those in the PRAM model.

```
parallel_for(int i = 0; i < n; ++i) {
    foo();
}
```

Example: Matrix Transpose

Example: In-place matrix transpose




```
for(int i = 1; i < n; ++i) {
  for(int j = 0; j < i; ++j) {
    double temp = A[i][j];
    A[i][j] = A[j][i];
    A[j][i] = temp;
  }
}
```

Example: Matrix Transpose

Example: In-place matrix transpose

```
parallel_for(int i = 1; i < n; ++i) {
  for(int j = 0; j < i; ++j) {
    double temp = A[i][j];
    A[i][j] = A[j][i];
    A[j][i] = temp;
  }
}
```

Assuming loop iterations execute in parallel

What is the work and span (in PRAM and the Cilk model)?

Example: Matrix Transpose

Example: In-place matrix transpose

```
parallel_for(int i = 1; i < n; ++i) {
  for(int j = 0; j < i; ++j) {
    double temp = A[i][j];
    A[i][j] = A[j][i];
    A[j][i] = temp;
  }
}
```

Assuming loop iterations execute in parallel

Work: $T_1(n) = \Theta(n^2)$

Span: $T_\infty(n) = \Theta(n)$

Parallelism: $T_1(n)/T_\infty(n) = \Theta(n)$

$\Theta(n + \lg n) = \Theta(n)$ **in the Cilk model**

Example: Matrix Transpose

Example: In-place matrix transpose

```
parallel_for(int i = 1; i < n; ++i) {
  parallel_for(int j = 0; j < i; ++j) {
    double temp = A[i][j];
    A[i][j] = A[j][i];
    A[j][i] = temp;
  }
}
```

Assuming loop iterations execute in parallel

What is the work and span (in PRAM and the Cilk model)?

Example: Matrix Transpose

Example: In-place matrix transpose

```
parallel_for(int i = 1; i < n; ++i) {
  parallel_for(int j = 0; j < i; ++j) {
    double temp = A[i][j];
    A[i][j] = A[j][i];
    A[j][i] = temp;
  }
}
```

Assuming loop iterations execute in parallel

Work: $T_1(n) = \Theta(n^2)$

Span: $T_\infty(n) = \Theta(1)$ **in PRAM,** $T_\infty(n) = \Theta(\lg n)$ **in Cilk**

Parallelism: $T_1(n)/T_\infty(n) = \Theta(n^2)$ **in PRAM,** $T_1(n)/T_\infty(n) = \Theta(n^2/\lg n)$ **in Cilk**

Quiz on Parallel Loops

(in PRAM)

```
parallel_for(int i = 0; i < n; i += 32) {
  for(int j = i; j < min(i+32, n); ++j) {
    A[j] += B[j];
  }
}
```

Work: $T_1 = \Theta(n)$

Span: $T_\infty = \Theta(1)$

Parallelism: $T_1/T_\infty = \Theta(n)$

```
parallel_for(int i = 0; i < n; i += n/P) {
  for(int j = i; j < min(i+n/P, n); ++j) {
    A[j] += B[j];
  }
}
```

Work: $T_1 = \Theta(n)$

Span: $T_\infty = \Theta(n/P)$

Parallelism: $T_1/T_\infty = \Theta(P)$

Quiz on Parallel Loops

(in PRAM)

```
parallel_for(int i = 0; i < n; i += 32) {
  for(int j = i; j < min(i+32, n); ++j) {
    A[j] += B[j];
  }
}
```

Work: $T_1 = \Theta(n)$

Span: $T_\infty = \Theta(1)$

Parallelism: $T_1/T_\infty = \Theta(n)$

```
parallel_for(int i = 0; i < n; i += n/P) {
  for(int j = i; j < min(i+n/P, n); ++j) {
    A[j] += B[j];
  }
}
```

Work: $T_1 = \Theta(n)$

Span: $T_\infty = \Theta(n/P)$

Parallelism: $T_1/T_\infty = \Theta(P)$

Too low! We want $T_1/T_\infty \gg P$

Three Performance Tips

1. Minimize the span to maximize parallelism. Try to generate 10 times more parallelism than processors for near-perfect linear speedup.

2. If you have plenty of parallelism, try to trade some of it off to reduce work overhead.

3. Use divide-and-conquer recursion or parallel loops rather than spawning one small thing after another.


```
parallel_for(int i = 0; i < n; ++i) {
  foo(i);
}
```


```
for(int i = 0; i < n; ++i) {
  spawn foo(i);
}
sync;
```

And Three More

4. Ensure that work/#spawns is sufficiently large.
   - Coarsen by using function calls and inlining near the leaves of recursion, rather than spawning.

5. Parallelize outer loops, as opposed to inner loops, if you're forced to make a choice.

6. Watch out for scheduling overheads.


```
parallel_for(int i = 0; i < 2; ++i) {
  for (int j = 0; j < n; ++j) {
    f(i, j);
  }
}
```


```
for (int j = 0; j < n; ++j) {
  parallel_for (int i = 0; i < 2; ++i) {
    f(i, j);
  }
}
```

Matrix Multiplication

Square-Matrix Multiplication

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

C A B

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Assume for simplicity that $n = 2^k$.

Parallelizing Matrix Multiply

```
parallel_for(int i = 0; i < n; ++i) {
  parallel_for(int j = 0; j < n; ++j) {
    for(int k = 0; k < n; ++k) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Work: $T_1(n) = \Theta(n^3)$

Span: $T_\infty(n) = \Theta(n)$

Parallelism: $T_1(n)/T_\infty(n) = \Theta(n^2)$

For 1000 x 1000 matrices, parallelism ~ 10^6

Recursive Matrix Multiplication

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{pmatrix} + \begin{pmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{pmatrix}$$

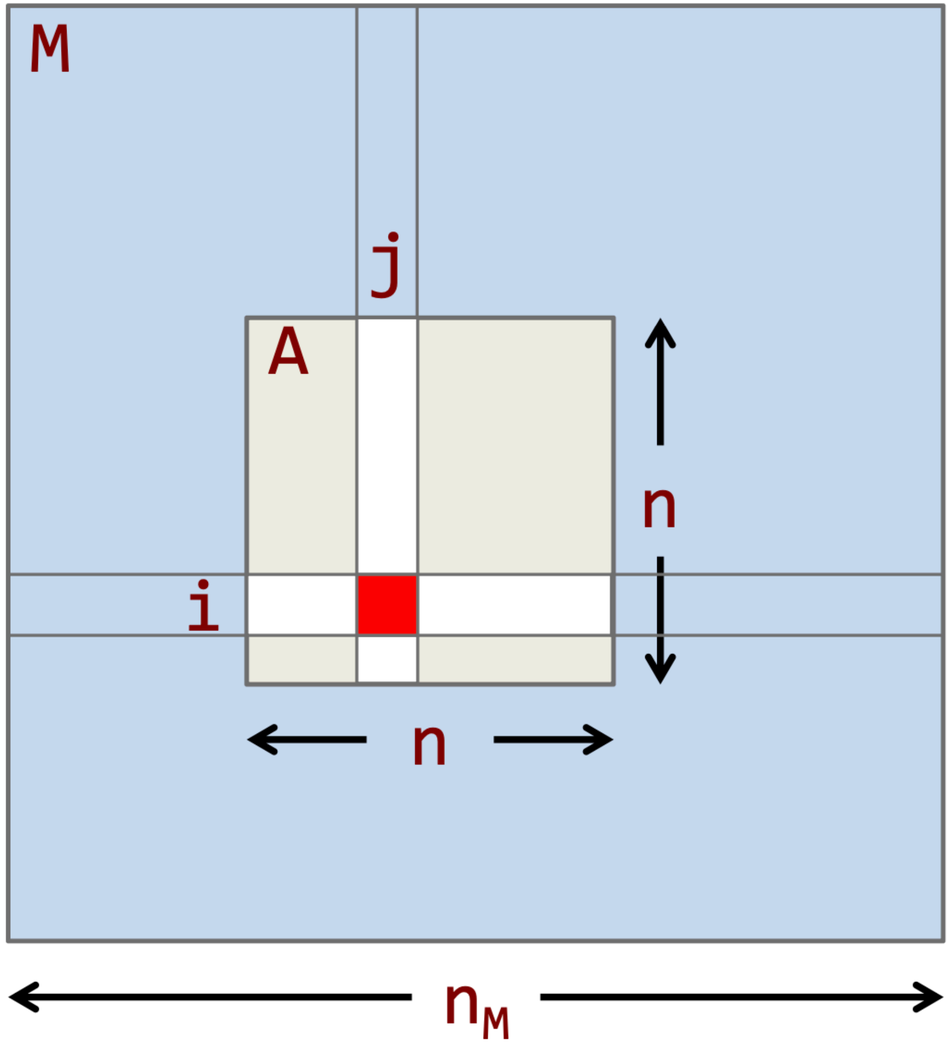
8 multiplications of $n/2 \times n/2$ matrices.
1 addition of $n \times n$ matrices.

Recall: Row-Major Layout

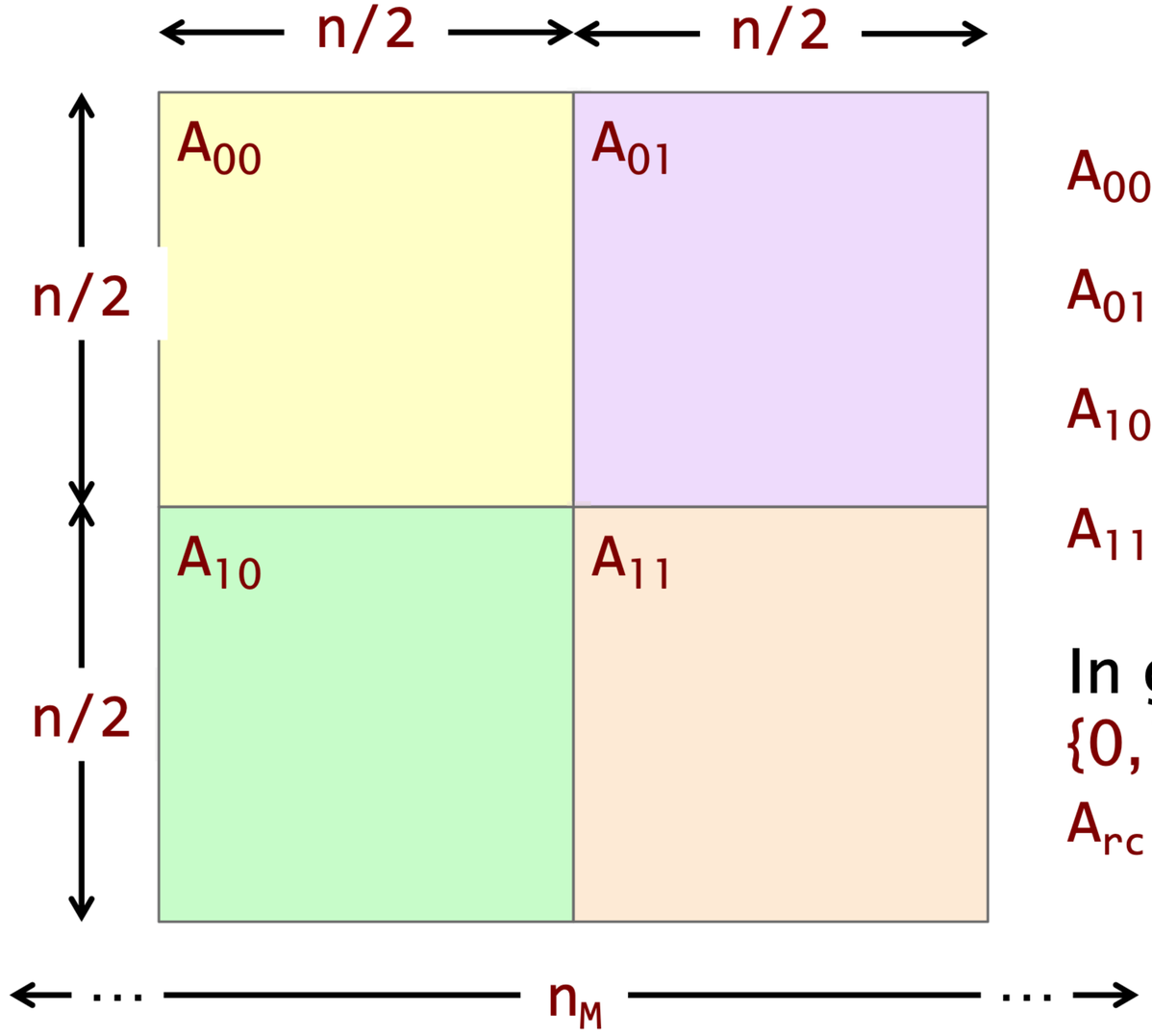
Row-major layout

If `A` is an $n \times n$ submatrix of an underlying matrix `M` with row size n_M, then the `(i,j)` element of `A` is `A[`n_M`*i + j]`.

Note: The dimension `n` does not enter into the calculation, although it does matter for bounds checking of `i` and `j`.



Recall: Divide-and-Conquer Matrices



$A_{00} = A$

$A_{01} = A + (n/2)$

$A_{10} = A + n_M*(n/2)$

$A_{11} = A + (n_M+1)*(n/2)$

In general, for $r, c \in \{0,1\}$, we have

$A_{rc} = A + (r*n_M+c)*(n/2)$

D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
                double *restrict A, int n_A,
                double *restrict B, int n_B,
                int n) {
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2))
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2))
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

The compiler can assume that the input matrices are not aliased

D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
                double *restrict A, int n_A,
                double *restrict B, int n_B,
                int n) {
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2))
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Row size of the underlying matrices

D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
                double *restrict A, int n_A,
                double *restrict B, int n_B,
                int n) {
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A,
  } else {
    double *D = malloc(
    assert(D != NULL);
#define n_D n
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2))
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

The three input matrices are n x n

D&C Matrix Multiplication

```
void mm_dac(double *res
                double *
                double *
                int n) {
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D                    f(*D));
    assert(D
#define n_D n
#define X(M,                     + c) * (n/2))
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Coarsen the leaves of the recursion to lower the overhead

The function adds the matrix product A*B to C

D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,

  if (n <= THRE
    mm_base(C,                          );
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M, r,                        ) * (n/2))
    parallel_spa                        , X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spa                        , X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spa                        , X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spa                        , X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Allocate a temporary n x n array D

The temporary array D has underlying row size n

D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *
    assert(D
#define n_D n
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2))
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Macro to compute submatrix indices

The C preprocessor's token-pasting operator

From MIT 6.172

D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2))
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
    m_add(C, n_C, D, n_
    free(D);
  }
}
```

Perform the 8 multiplications of (n/2) x (n/2) submatrices recursively in parallel

Wait for all spawned subcomputations to finish

D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
                double *restrict A, int n_A,
                double *restrict B, int n_B,
                int n) {
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M, r, c) (M + r*(n_ ## M)     c) * (n/2))
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Add the temporary matrix D into output matrix C

D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
                double *restrict A, int n_A,
                double *restrict B, int n_B,
                int n) {
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M, r, c) (M + r*(n_ ## M) * (n/2)) + (c) * (n/2))
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X
    parallel_spawn mm_dac(X(D,0,0), n_D, X
    parallel_spawn mm_dac(X(D,0,1), n_D, X
    parallel_spawn mm_dac(X(D,1,0), n_D, X
    parallel_spawn mm_dac(X(D,1,1), n_D, X
    parallel_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Add the temporary matrix D into output matrix C

```
void m_add (double *restrict C, int n_C,
                double *restrict D, int n_D,
                int n) {
  parallel_for(int i = 0; i < n; ++i) {
    parallel_for(int j = 0; j < n; ++j) {
      C[i*n_C + j] += D[i*n_D + j];
    }
  }
}
```

D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
                double *restrict A, int n_A,
                double *restrict B, int n_B,
                int n) {
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2))
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Clean up and return

Analysis of Matrix Addition

```
void m_add (double *restrict C, int n_C,
            double *restrict D, int n_D,
            int n) {
  parallel_for(int i = 0; i < n; ++i) {
    parallel_for(int j = 0; j < n; ++j) {
      C[i*n_C + j] += D[i*n_D + j];
    }
  }
}
```

Work: $A_1(n) = \Theta(n^2)$

Span: $A_\infty(n) = \Theta(\lg n)$

Cilk model

Work of Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
                double *restrict A, int n_A,
                double *restrict B, int n_B,
                int n) {
// base case up here
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,
    parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,
    parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,
    parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B
    parallel_spawn mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(
    parallel_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
```

Case 1 of Master Method

$$n^{\log_b a} = n^{\log_2 8} = n^3$$

$$f(n) = \Theta(n^2)$$

Work: $M_1(n) = 8M_1(n/2) + A_1(n) + \Theta(1)$

$= 8M_1(n/2) + \Theta(n^2)$

$= \Theta(n^3)$

Span of Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
  base case up here
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
```

Maximum over spawns

Case 2 of Master Method

$$n^{\log_b a} = n^{\log_2 1} = 1$$

$$f(n) = \Theta(n^{\log_b a} \lg^1 n)$$

Span: $M_\infty(n) = M_\infty(n/2) + A_\infty(n) + \Theta(1)$

$= M_\infty(n/2) + \Theta(\lg n)$

$= \Theta(\lg^2 n)$

Parallelism of Matrix Multiply

Work: $M_1(n) = \Theta(n^3)$

Span: $M_\infty(n) = \Theta(\lg^2 n)$

Cilk model

Parallelism: $M_1(n)/M_\infty(n) = \Theta(n^3/lg^2 n)$

For 1000 x 1000 matrices, parallelism ~ $(10^3)^3/10^2 = 10^7$

From MIT 6.172

D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
                double *restrict A, int n_A,
                double *restrict B, int n_B,
                int n) {
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
#define n_D n
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2))
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    parallel_
    parallel_
    parallel_
    parallel_
    parallel_
    m_add(C,
    free(D);
  }
}
```

Idea: Since minimizing storage tends to yield higher performance, **trade some of the ample parallelism** for less storage.

No-Temp D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
                double *restrict A, int n_A,
                double *restrict
                int n) {
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B,
  } else {
#define X(M, r, c) (M + r*(n      M) + c) * (n/2))
    parallel_spawn mm_dac(      ,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_da  X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_    ac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn m      dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    parallel_sync;

    parallel_spawn mm_dac(X(C,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
  }
}
```

Do 4 subproblems, sync, then do 4 more subproblems

No-Temp D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2))
    parallel_spawn mm_dac(X(C,0,0), n                A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n                A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n                A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n                A, X(B,0,1), n_B, n/2);
    parallel_sync;

    parallel_spawn mm_dac(X(C,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
  }
}
```

Reuse C without racing

Saves space, but at what expense?

No-Temp D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
                double *restrict A, int n_A,
                double *restrict B, int n_B,
                int n) {
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2))
    parallel_spawn mm_dac(X(C,0,0), n                A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n                A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n                A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n                A, X(B,0,1), n_B, n/2);
    parallel_sync;

    parallel_spawn mm_dac(X(C,0,0), n_D, X(A,0,1), n_
    parallel_spawn mm_dac(X(C,0,1), n_D, X(A,0,1), n_
    parallel_spawn mm_dac(X(C,1,0), n_D, X(A,1,1), n_
    parallel_spawn mm_dac(X(C,1,1), n_D, X(A,1,1), n_
    parallel_sync;
  }
}
```

Reuse C without racing

Case 1 of Master Method

$$n^{\log_b a} = n^{\log_2 8} = n^3$$

$$f(n) = \Theta(1)$$

Work: $M_1(n) = 8M_1(n/2) + \Theta(1) = \Theta(n^3)$

No-Temp D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
                double *restrict A, int n_A,
                double *restrict B, int n_B,
                int n) {
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2))
    parallel_spawn mm_dac(X(C,0,0), n                    A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n                    A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n                    A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n                    A, X(B,0,1), n_B, n/2);
    parallel_sync;

    parallel_spawn mm_dac(X(C,0,0), n_D, X(A,0,1), n_
    parallel_spawn mm_dac(X(C,0,1), n_D, X(A,0,1), n_
    parallel_spawn mm_dac(X(C,1,0), n_D, X(A,1,1), n_
    parallel_spawn mm_dac(X(C,1,1), n_D, X(A,1,1), n_
    parallel_sync;
  }
}
```

Reuse C without racing

Case 1 of Master Method

$$n^{\log_b a} = n^{\log_2 2} = n$$

$$f(n) = \Theta(1)$$

Span: $M_\infty(n) = 2M_\infty(n/2) + \Theta(1) = \Theta(n)$

Parallelism of No-Temp Multiply

Work: $M_1(n) = \Theta(n^3)$

Span: $M_\infty(n) = \Theta(n)$

Higher than with temp

Parallelism: $M_1(n)/M_\infty(n) = \Theta(n^2)$

For 1000 x 1000 matrices, parallelism ~ $(10^3)^2 = 10^6$

No-temp version is faster in practice

Merge Sort

Merging Two Sorted Arrays

```
void merge(int *C, int *A, int na, int *B, int nb) {
  while (na > 0 && nb > 0) {
    if (*A <= *B) {
      *C++ = *A++, na--;
    } else {
      *C++ = *B++, nb–;
    }
  }
  while (na > 0) {
    *C++ = *A++, na--;
  }
  while (nb > 0) {
    *C++ = *B++, nb–;
  }
}
```

Work to merge n elements = $\Theta(n)$

Merge Sort

```
void merge_sort(int *B, int *A, int n) {
  if (n == 1) {
    B[0] = A[0];
  } else {
    int C[n];
    parallel_spawn merge_sort(C, A, n/2);
    parallel_spawn merge_sort(C+n/2, A+n/2, n-n/2);
    parallel_sync;
    merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```

Work: $T_1(n) = 2T_1(n/2) + \Theta(n)$

$= \Theta(n \lg n)$

Work of Merge Sort

```
void merge_sort(int *B, int *A, int n) {
  if (n == 1) {
    B[0] = A[0];
  } else {
    int C[n];
    parallel_spawn merge_sort(C, A, n/2);
    parallel_spawn merge_sort(C+n/2, A+n/2, n-n/2);
    parallel_sync;
    merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```

Case 2 of Master Method

$$n^{\log_b a} = n^{\log_2 2} = n$$

$$f(n) = \Theta(n^{\log_b a} \lg^0 n)$$

Work: $T_1(n) = 2T_1(n/2) + \Theta(n)$

$$= \Theta(n \lg n)$$

From MIT 6.172

Span of Merge Sort

```
void merge_sort(int *B, int *A, int n) {
  if (n == 1) {
    B[0] = A[0];
  } else {
    int C[n];
    parallel_spawn merge_sort(C, A, n/2);
    parallel_spawn merge_sort(C+n/2, A+n/2, n-n/2);
    parallel_sync;
    merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```

Case 3 of Master Method

$$n^{\log_b a} = n^{\log_2 1} = 1$$

$$f(n) = \Theta(n)$$

Span: $T_\infty(n) = T_\infty(n/2) + \Theta(n)$

$= \Theta(n)$

Parallelism of Merge Sort

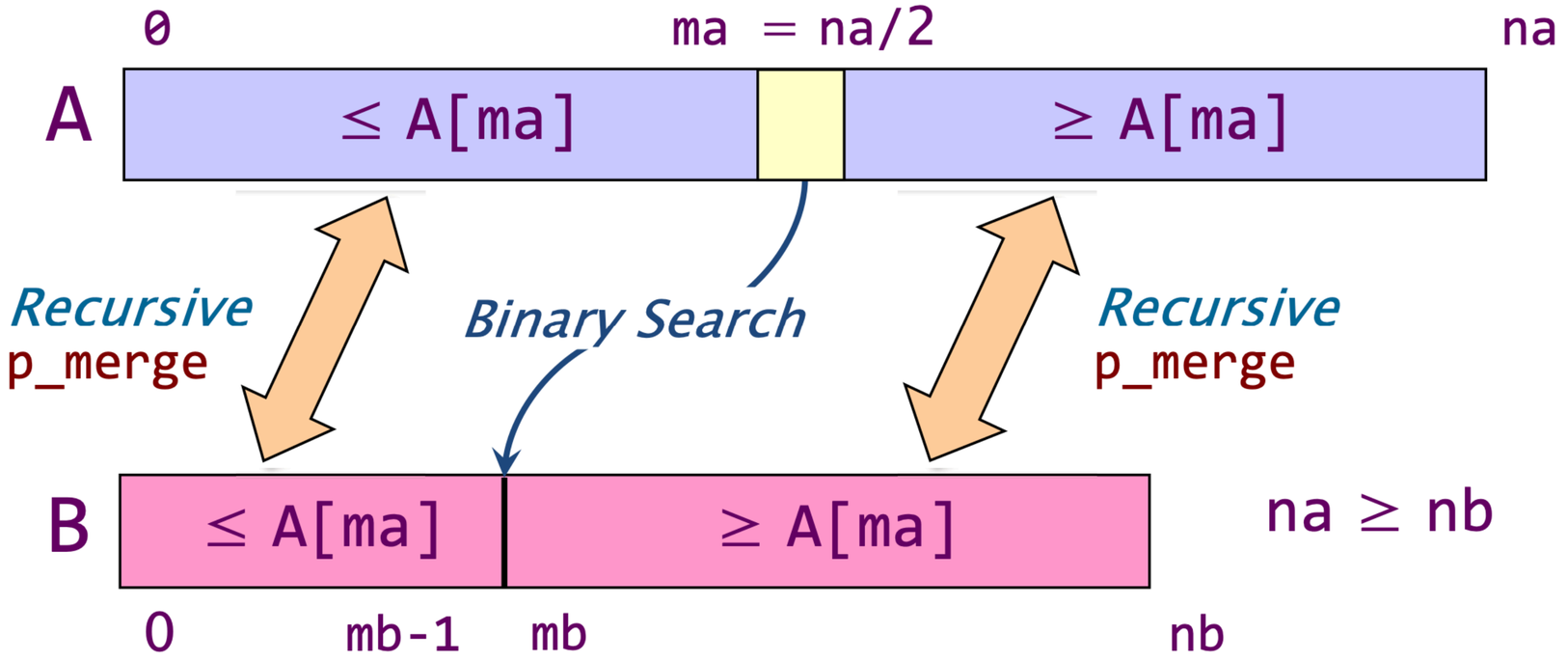
Work: $T_1(n) = \Theta(n \lg n)$

Span: $T_\infty(n) = \Theta(n)$

Parallelism: $T_1(n)/T_\infty(n) = \Theta(\lg n)$

We need to parallelize the merge!

Parallel Merge



KEY IDEA: If the total number of elements to be merged in the two arrays is $n = na + nb$, the total number of elements in the larger of the two recursive merges is at most $(3/4)n$.

Parallel Merge

```
void p_merge(int *C, int *A, int na, int *B, int nb) {
  if (na < nb) {
    p_merge(C, B, nb, A, na); // flip A and B
  } else if (na == 0) {
    return;
  } else {
    int ma = na/2;
    int mb = binary_search(A[ma], B, nb);
    C[ma+mb] = A[ma];
    parallel_spawn p_merge(C, A, ma, B, mb);
    parallel_spawn p_merge(C+ma+mb+1, A+ma+1, na-ma-1, B+mb, nb-mb);
    parallel_sync;
  }
}
```

Coarsen base cases for efficiency.

Span of Parallel Merge

```
void p_merge(int *C, int *A, int na, int *B, int nb) {
  if (na < nb) {
    p_merge(C, B, nb, A, na); // flip A and B
  } else if (na == 0) {
    return;
  } else {
    int ma = na/2;
    int mb = binary_search(A[ma], B, nb);
    C[ma+mb] = A[ma];
    parallel_spawn p_merge(C, A, ma, B, mb);
    parallel_spawn p_merge(C+ma+mb+1, A+ma+1, n
    parallel_sync;
  }
}
```

Case 2 of Master Method

$$n^{\log_b a} = n^{\log_{4/3} 1} = 1$$

$$f(n) = \Theta(n^{\log_b a} \lg^1 n)$$

Span: $T_\infty(n) = T_\infty(3n/4) + \Theta(\lg n)$

$= \Theta(\lg^2 n)$

Work of Parallel Merge

```
void p_merge(int *C, int *A, int na, int *B, int nb) {
  if (na < nb) {
    p_merge(C, B, nb, A, na); // flip A and B
  } else if (na == 0) {
    return;
  } else {
    int ma = na/2;
    int mb = binary_search(A[ma], B, nb);
    C[ma+mb] = A[ma];
    parallel_spawn p_merge(C, A, ma, B, mb);
    parallel_spawn p_merge(C+ma+mb+1, A
    parallel_sync;
  }
}
```

-mb);

Complicated / not in Master Method cases

Span: $T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n)$,

where $1/4 \leq \alpha \leq 3/4$

Claim: $T_1(n) = \Theta(n)$

From MIT 6.172

Analysis of Work Recurrence

$$T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n),$$
where $1/4 \le \alpha \le 3/4$.

Substitution method: Inductive hypothesis is $T_1(k) \le c_1 k - c_2 \lg k$, where $c_1, c_2 > 0$. Prove that the relation holds, and solve for c_1 and c_2.

$$\begin{aligned}
T_1(n) &\le c_1(\alpha n) - c_2\lg(\alpha n) + c_1(1-\alpha)n - c_2\lg((1-\alpha)n) + \Theta(\lg n) \\
&\le c_1 n - c_2\lg(\alpha n) - c_2\lg((1-\alpha)n) + \Theta(\lg n) \\
&\le c_1 n - c_2\,(\lg(\alpha(1-\alpha)) + 2\lg n\,) + \Theta(\lg n) \\
&\le c_1 n - c_2\lg n - (c_2(\lg n + \lg(\alpha(1-\alpha))) - \Theta(\lg n)) \\
&\le c_1 n - c_2\lg n,
\end{aligned}$$
by choosing c_2 large enough. Choose c_1 large enough to handle the base case.

Parallelism of Parallel Merge

Work: $T_1(n) = \Theta(n)$

Span: $T_\infty(n) = \Theta(\lg^2 n)$

Parallelism: $T_1(n)/T_\infty(n) = \Theta(n/\lg^2 n)$

Parallel Merge Sort

```
void p_merge_sort(int *B, int *A, int n) {
  if (n == 1) {
    B[0] = A[0];
  } else {
    int C[n];
    parallel_spawn merge_sort(C, A, n/2);
    parallel_spawn merge_sort(C+n/2, A+n/2, n-n/2);
    parallel_sync;
    p_merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```

Same as before except with parallel merge

Case 2 of Master Method

$$n^{\log_b a} = n^{\log_2 2} = n$$

$$f(n) = \Theta(n^{\log_b a} \lg^0 n)$$

Work: $T_1(n) = 2T_1(n/2) + \Theta(n)$

$= \Theta(n \lg n)$

Parallel Merge Sort

```
void p_merge_sort(int *B, int *A, int n) {
  if (n == 1) {
    B[0] = A[0];
  } else {
    int C[n];
    parallel_spawn merge_sort(C, A, n/2);
    parallel_spawn merge_sort(C+n/2, A+n/2, n-n/2);
    parallel_sync;
    p_merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```

Same as before except with parallel merge

Case 2 of Master Method

$$n^{\log_b a} = n^{\log_2 1} = 1$$

$$f(n) = \Theta(n^{\log_b a} \lg^2 n)$$

Span: $T_\infty(n) = T_\infty(n/2) + \Theta(\lg^2 n)$

$= \Theta(\lg^3 n)$

Parallelism of Parallel Merge Sort

Work: $T_1(n) = \Theta(n \lg n)$

Span: $T_\infty(n) = \Theta(\lg^3 n)$

Parallelism: $T_1(n)/T_\infty(n) = \Theta(n/\lg^2 n)$