Algorithms at Scale

(Week 7)

Puzzle of the Day:

100 prisoners. Every so often, one is chosen at random to enter a room with a light bulb. You can turn the light bulb on or off.

- WIN if one prisoner announces correctly that all have visited the room.
- LOSE if announcement is incorrect.

What if, initially, the state of the light is unknown, either on or off?

Summary

Last Week: Clustering

k-median clustering

LP approximation algorithm

Streaming

Other clustering problems

Today: Caching

External memory model

- How to predict the performance of algorithms?

B-trees

- Efficient searching

Write-optimized data structures

- Buffer trees

Cache-oblivious algorithms

- van Emde Boas memory layout

Summary

Last Week: Clustering

k-median clustering

LP approximation algorithm

Streaming

Other clustering problems

Today: Caching

External memory model

- How to predict the performance of algorithms?

B-trees

- Efficient searching

Write-optimized data structures

- Buffer trees

Cache-oblivious algorithms

- van Emde Boas memory layout

MiniProjects

Four basic topics

1) Dimensionality reduction (i.e., sampling algorithms)

2) Streaming data analysis

3) Cache efficient search structures (e.g., log-structured merge trees, COLA)

4) Algorithms for the MPC / k-server model.

MiniProjects

Four basic topics

1) Dimensionality reduction (i.e., sampling algorithms)

2) Streaming data analysis

3) Cache efficient search structures (e.g., log-structured merge trees, COLA)

4) Algorithms for the MPC / k-server model.

Or choose your own....

MiniProjects

Three parts:

1) Explain:

Read research paper or other information on the topic, and write an explanatory paper that explains

2) Extend:

Implement the data structures described and run experiments, or design the algorithm that is requested.

3) Presentation:

Give a presentation on the topic.

Record and submit your presentation.

6 (or so) will be chosen to present in class in Week 13.

MiniProjects

This week:

1) Form a team of two.

Choose a partner with a shared interest.

I'll put up a spreadsheet to help do matching.

2) Choose a topic.

I'll post the four topics, along with some specific questions to answer.

3) Do background reading.

Find key material and begin to read it.

To submit: team, topic, summary of background reading.

Summary

Last Week: Clustering

k-median clustering

LP approximation algorithm

Streaming

Other clustering problems

Today: Caching

External memory model

- How to predict the performance of algorithms?

B-trees

- Efficient searching

Write-optimized data structures

- Buffer trees

Cache-oblivious algorithms

- van Emde Boas memory layout

Why do we analyze algorithms?

1. To ensure that it does the right thing (i.e., *correctness*).

2. To predict the *performance* (or determine which is fastest).

Predicting Performance

Example: 100 TB of data

1) Store data sorted in an array

- ⇒ Scan all the data: O(n)
- ⇒ (Binary) search: O(log n)

2) Store data in a linked list

- ⇒ Scan all the data: O(n)
- ⇒ Search: O(n)

3) Store data in a red-black tree

- ⇒ Scan all the data: O(n)
- ⇒ Search: O(log n)

Predicting Performance

Example: 100 TB of data

1) Store data sorted in an array

- ⇒ Scan all the data: O(n)
- ⇒ (Binary) search: O(log n)

2) Store data in a linked list

- ⇒ Scan all the data: O(n)
- ⇒ Search: O(n)

3) Store data in a red-black tree

- ⇒ Scan all the data: O(n)
- ⇒ Search: O(log n)

Same performance!

Predicting Performance

Example: 100 TB of data

1) Store data sorted in an array

⇒ Scan all the data: O(n)

⇒ (Binary) search: O(log n)

2) Store data in a linked list

⇒ Scan all the data: O(n)

⇒ Search: O(n)

3) Store data in a red-black tree

⇒ Scan all the data: O(n)

⇒ Search: O(log n)

Same performance!

Predicting Performance

Example: 100 TB of data

1) Store data sorted in an array

⇒ Scan all the data: O(n)

⇒ (Binary) search: O(log n)

2) Store data in a linked list

⇒ Scan all the data: O(n)

⇒ Search: O(n)

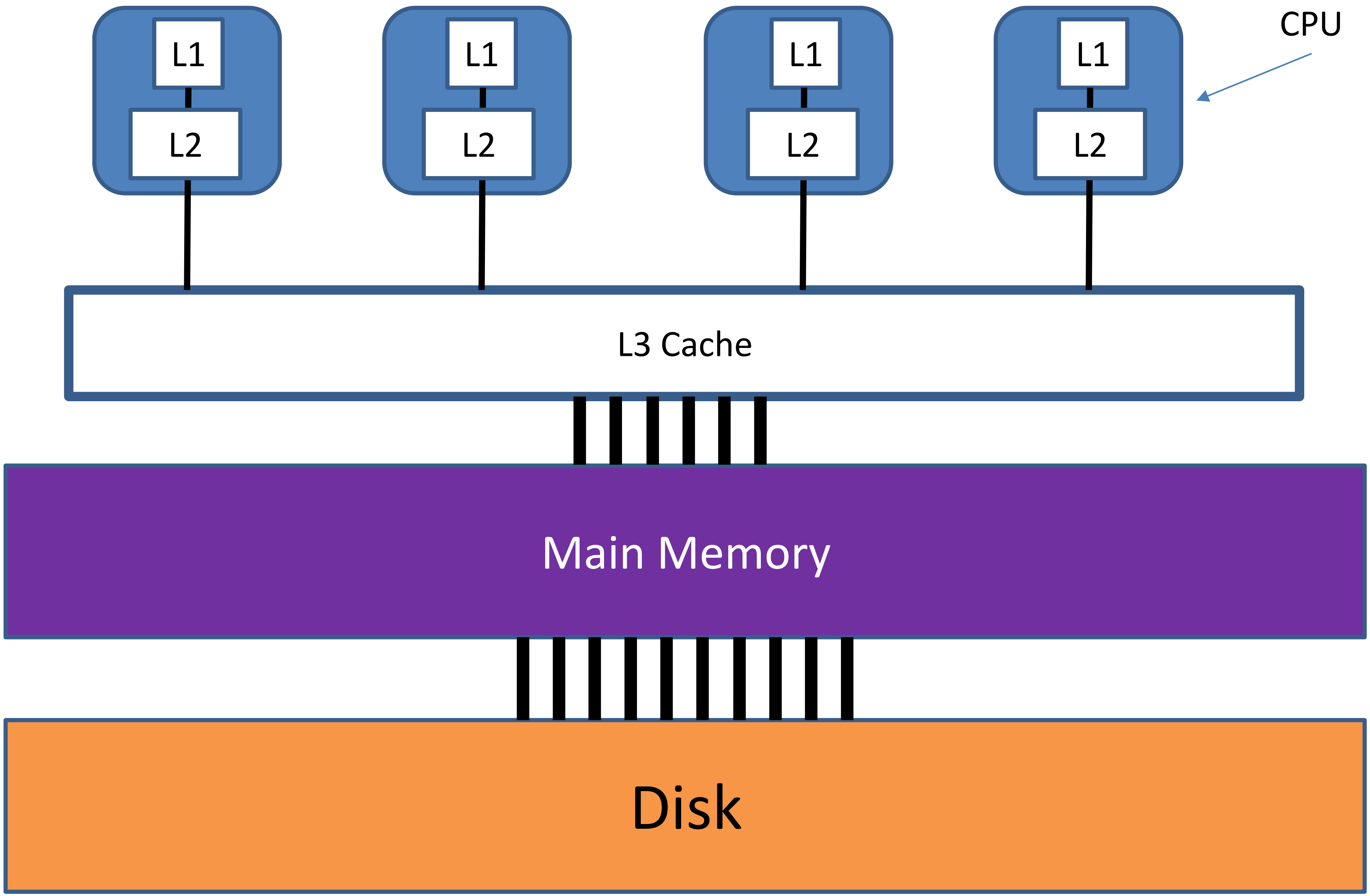
3) Store data in a red-black tree

⇒ Scan all the data: O(n)

⇒ Search: O(log n)

Analysis is not predicting performance very well!

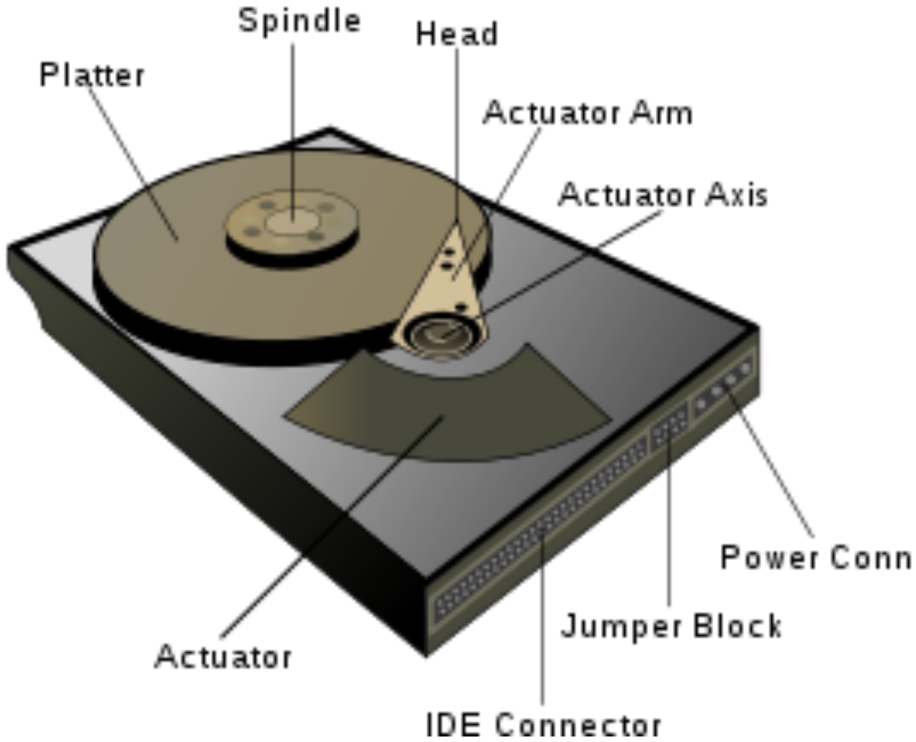
A Real Computer (?)



Disks

Where is most data stored? Hard disk!

- Magnetic
- Mechanical
- Slow (6000rpm = 10ms)



Two step access:

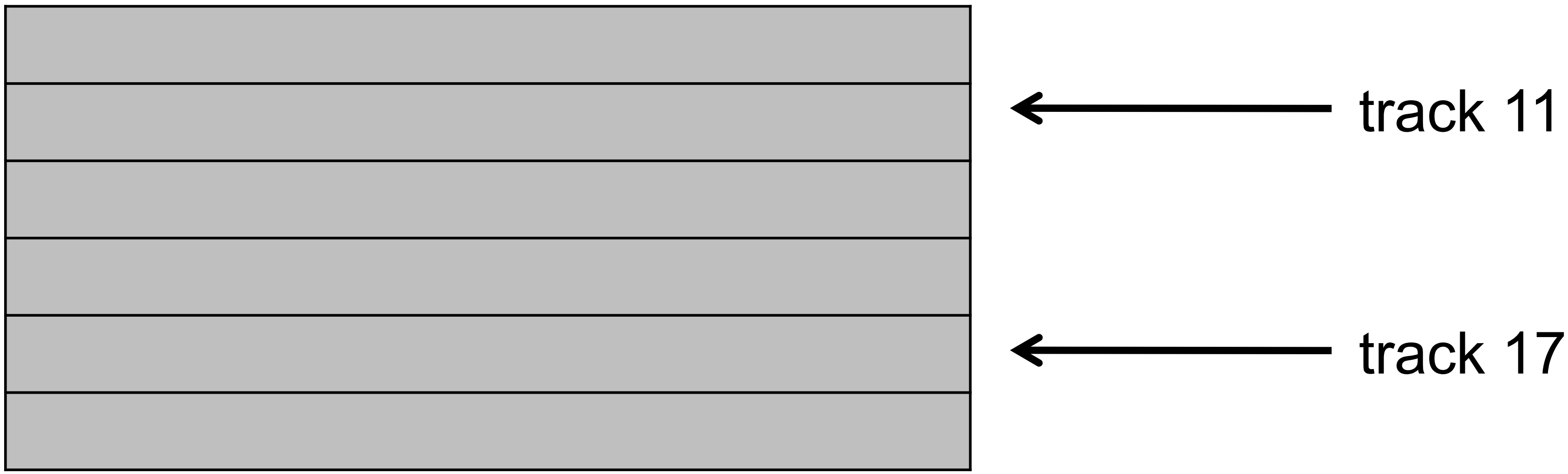
1. seek *(find right track)*
2. read track

Disks

Two step access:

1. seek *(find right track)*
2. read track

In practice: Cache entire track

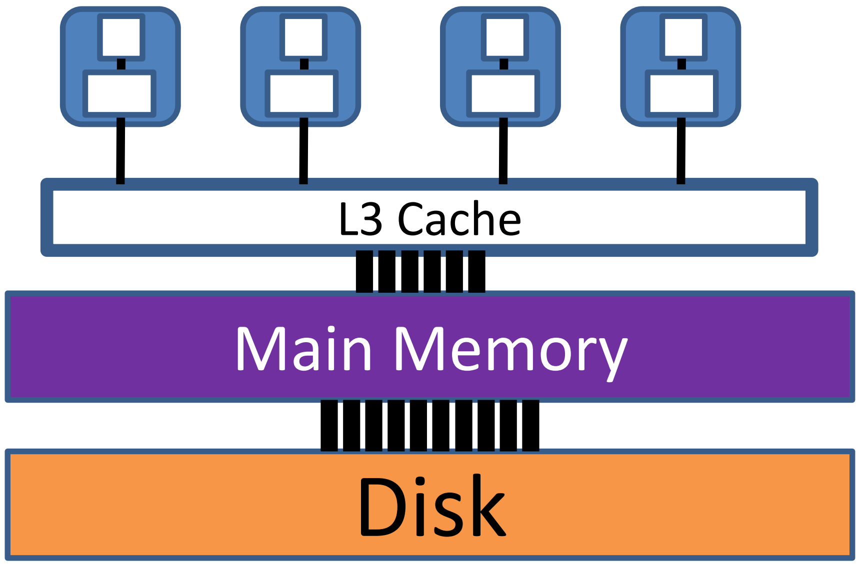


Haswell Architecture (2-18 cores)

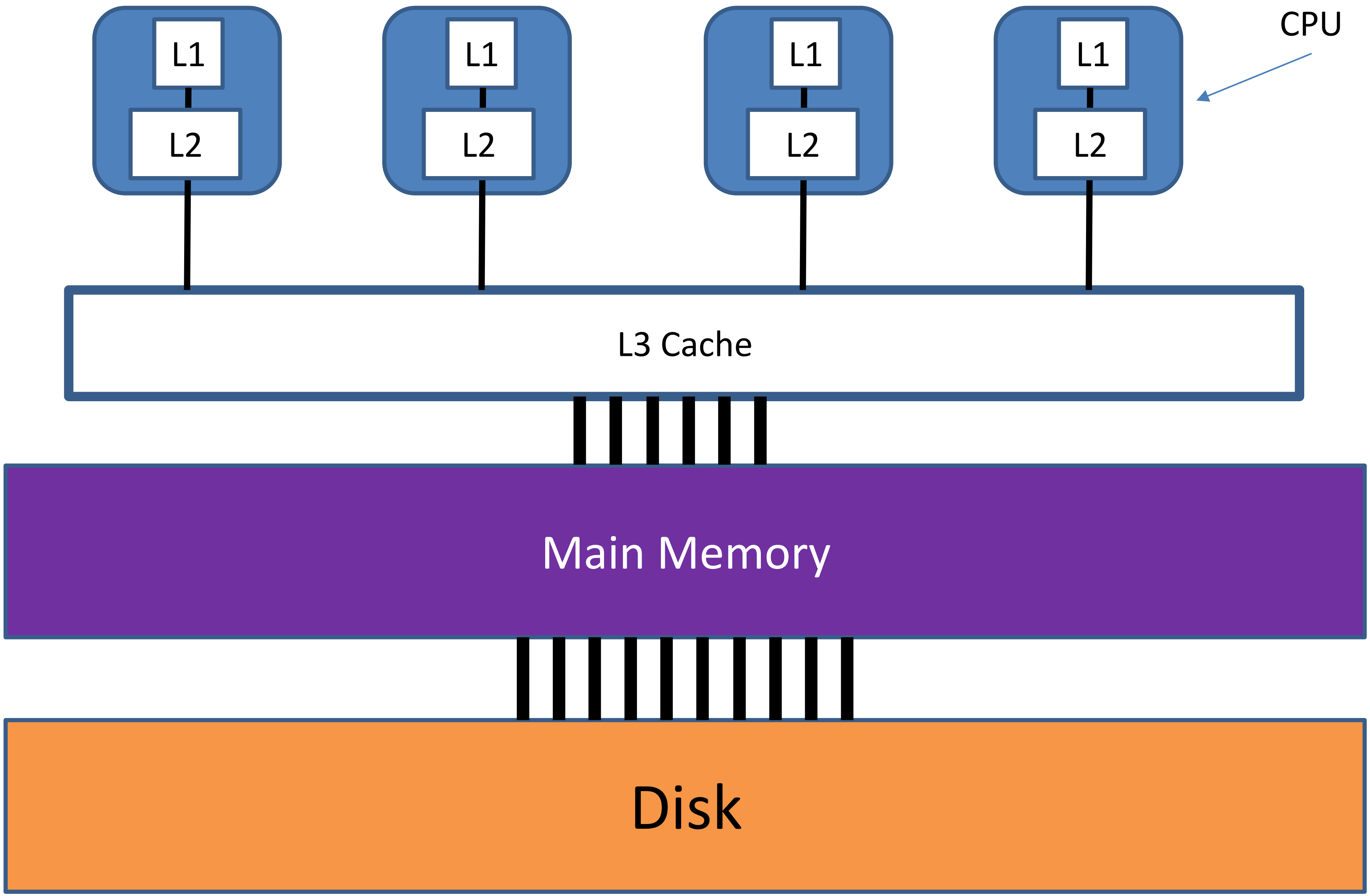
Memory Type	size	line size	clock cycles
L1 cache	64 KB	64 B	~4
L2 cache	256 KB	64 B	~10
L3 cache	2-40 MB	64 B	40-74
L4 (optional)	128 MB		
Main Memory	< 128 GB	16 KB	~200-350
SSD Disk	BIG	Variable (e.g., 16KB)	~20,000
Disk	BIGGER	Variable (e.g., 16KB)	~20,000,000

Notes:

- Several other ”caches” e.g., TLB, micro-op cache, instruction cache, etc.
- L1/L2 caches are per core.
- L3/L4 cache are shared per socket.
- Main memory shared cross socket.



A Real Computer (?)



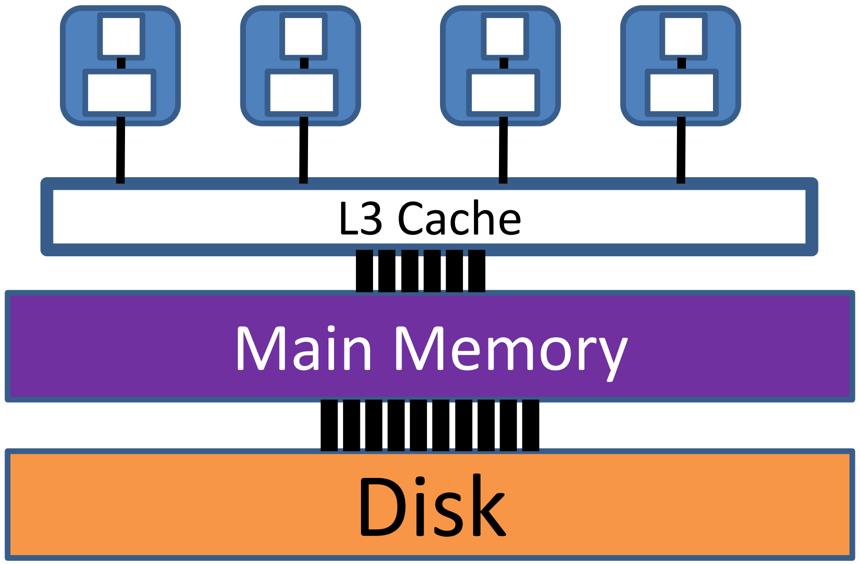
Haswell Architecture

A simple example calculation:

What fraction of operations ”hit” each cache?

- ⇒ 90% L1 hit rate (4 cycles)
- ⇒ 8% L2 hit rate (10 cycles)
- ⇒ 2% main memory (300 cycles)

Just an example..



Haswell Architecture

A simple example calculation:

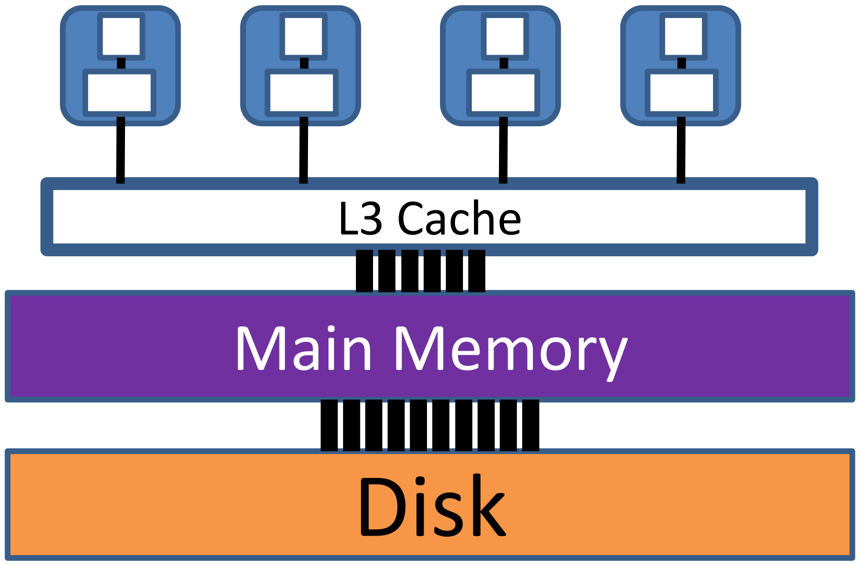
What fraction of operations ”hit” each cache?

- ⇒ 90% L1 hit rate (4 cycles)
- ⇒ 8% L2 hit rate (10 cycles)
- ⇒ 2% main memory (300 cycles)

Just an example..

What fraction of time for each cache?

- ⇒ 35% waiting for L1
- ⇒ 8% waiting for L2
- ⇒ 57% waiting for main memory



Conclusion:

98% cache hit ➔
57% waiting on main memory

Haswell Architecture

A simple example calculation:

What fraction of operations ”hit” each cache?

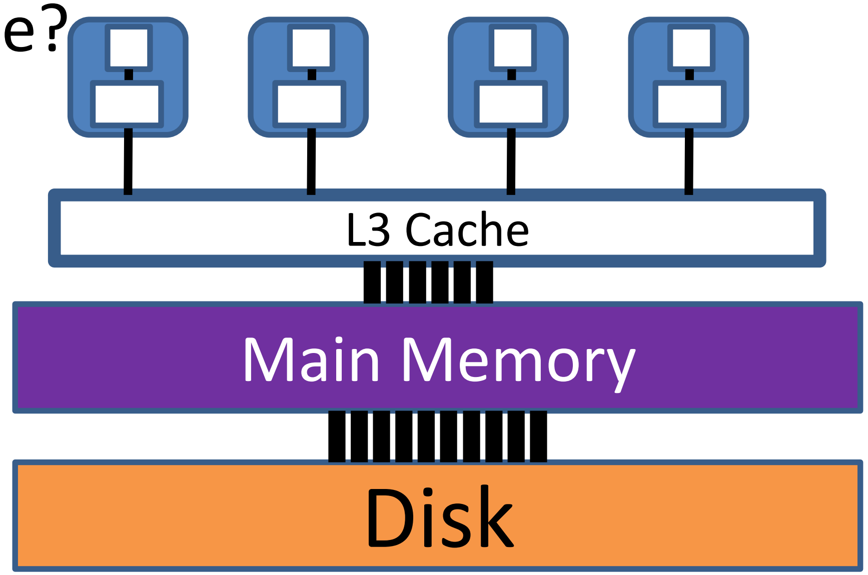
- ⇒ 90% L1 hit rate (4 cycles)
- ⇒ 8% L2 hit rate (10 cycles)
- ⇒ 1.8% main memory (300 cycles)
- ⇒ 0.2% disk (20,000,000 cycles)

Just an example..

What fraction of time for each cache?

- ⇒ 99.98% waiting for disk

Disk is much, much worse!



Where is the bottleneck?

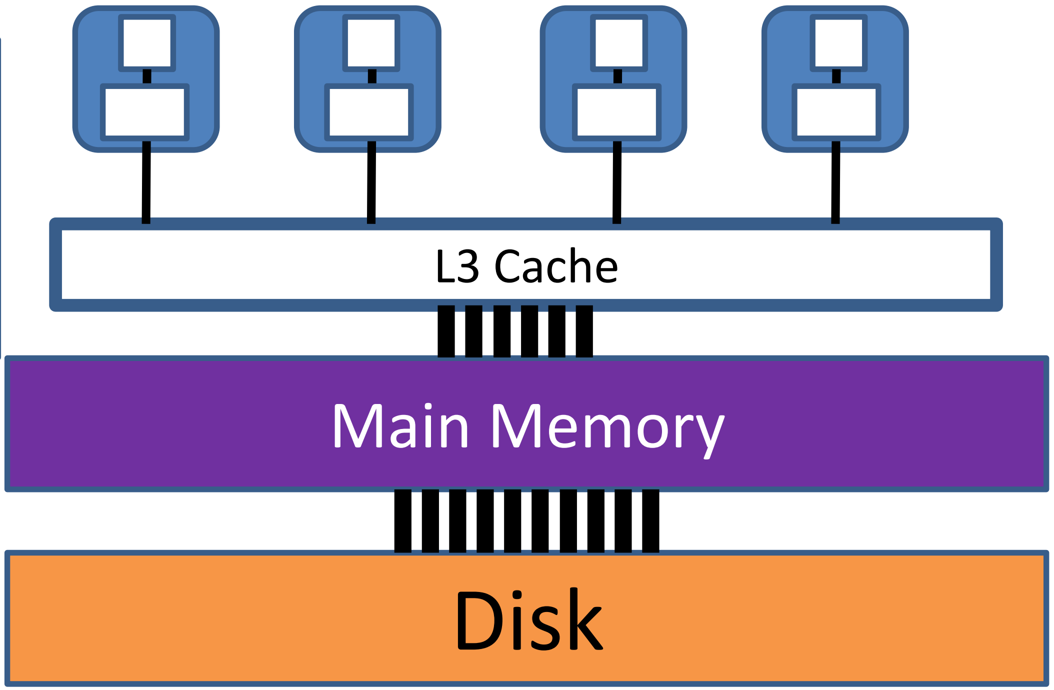
The bottleneck depends on the application:

- Small working set data lives in L1/L2 cache ➔ fast.
- Medium working set data lives in main memory ➔ bottleneck is memory latency.
- Big data lives on disk ➔ bottleneck is disk latency / bandwidth.

For most applications, one level dominates the cost.

(Costs grow fast!

Largest level dominates.)

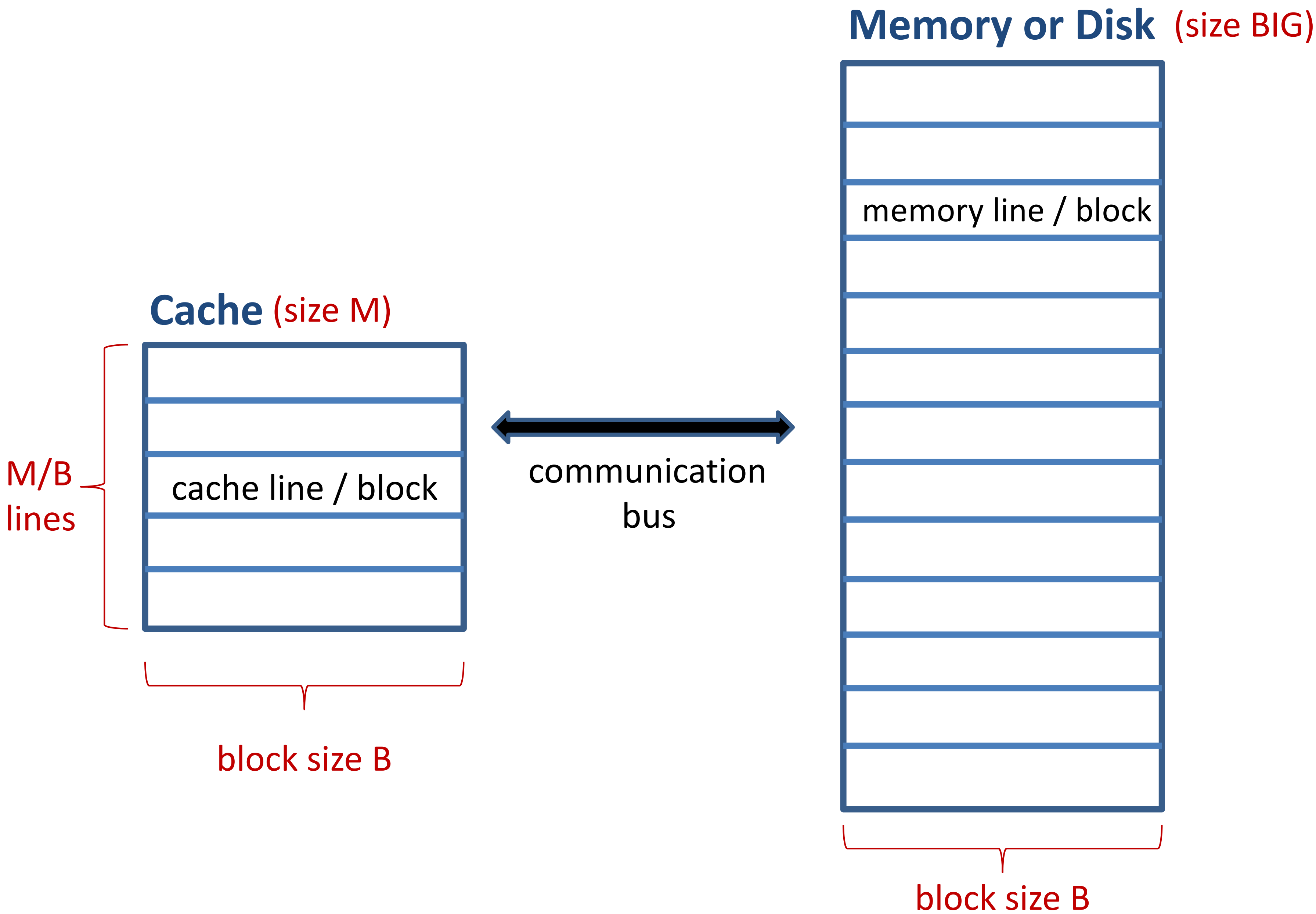


External Memory Model (Aggarwal, Vitter 1988)

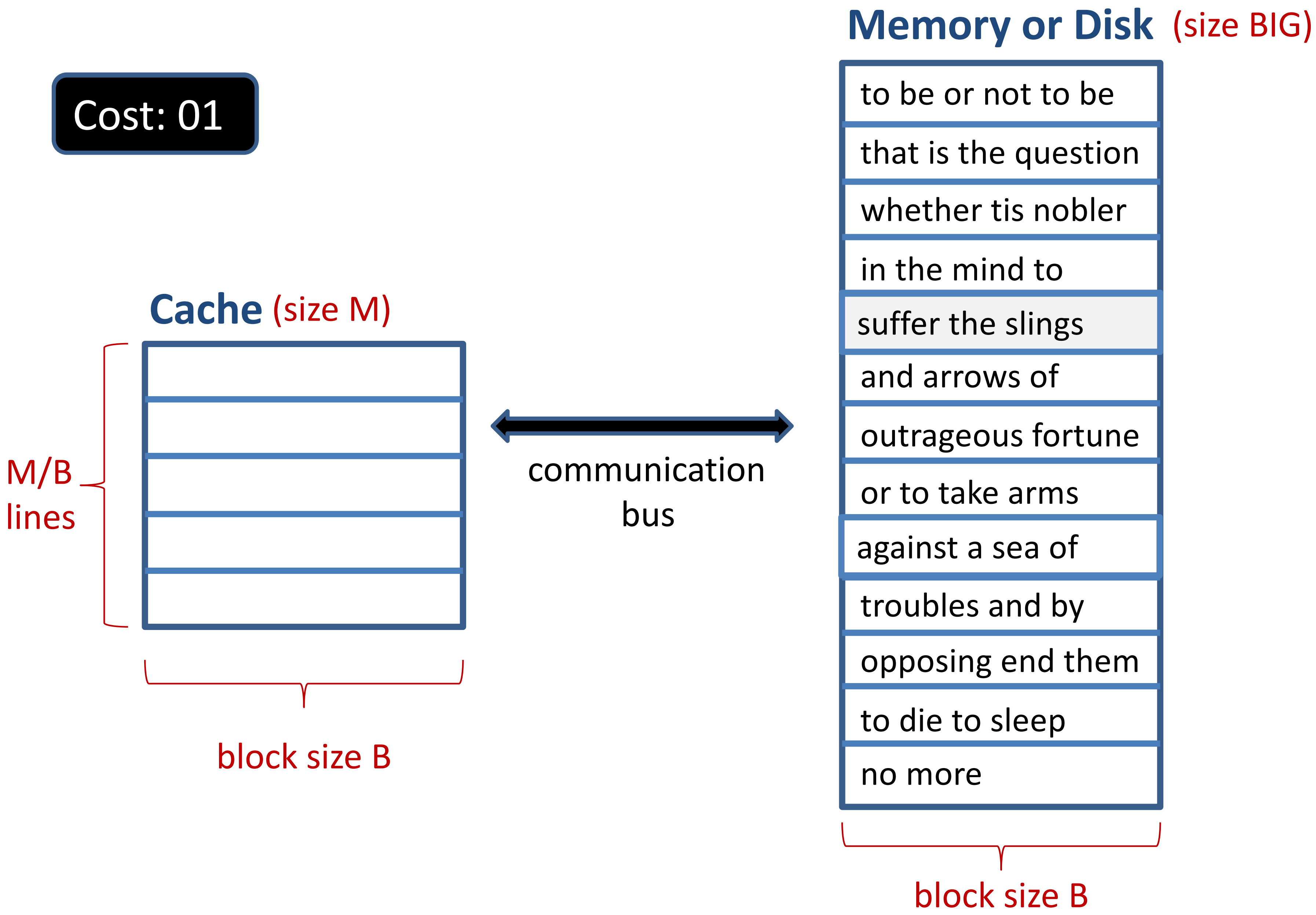
Goal:

- Simple model (i.e., *tractable*)
- Sufficiently accurate model (i.e., *useful*)

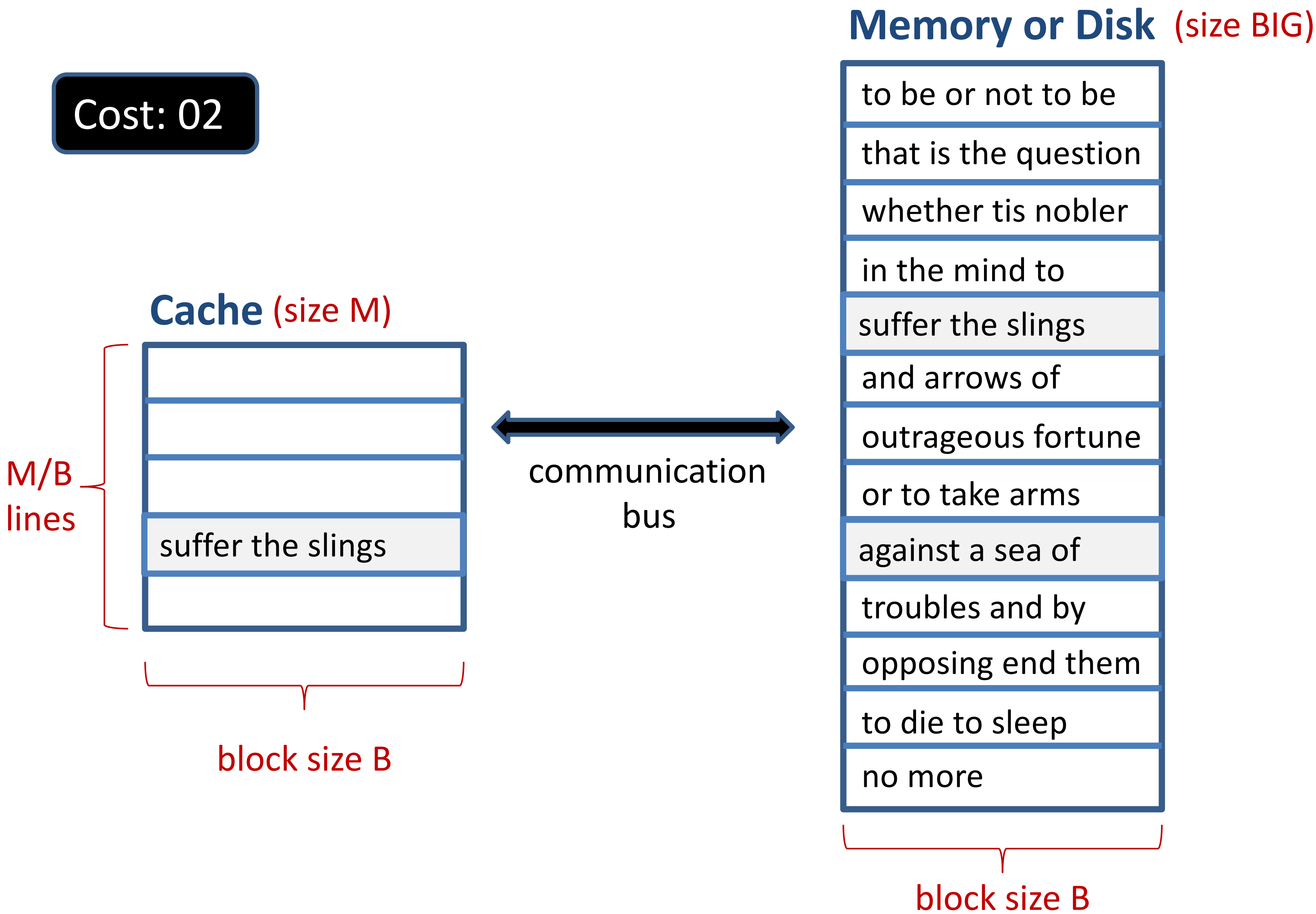
External Memory Model (Aggarwal, Vitter 1988)



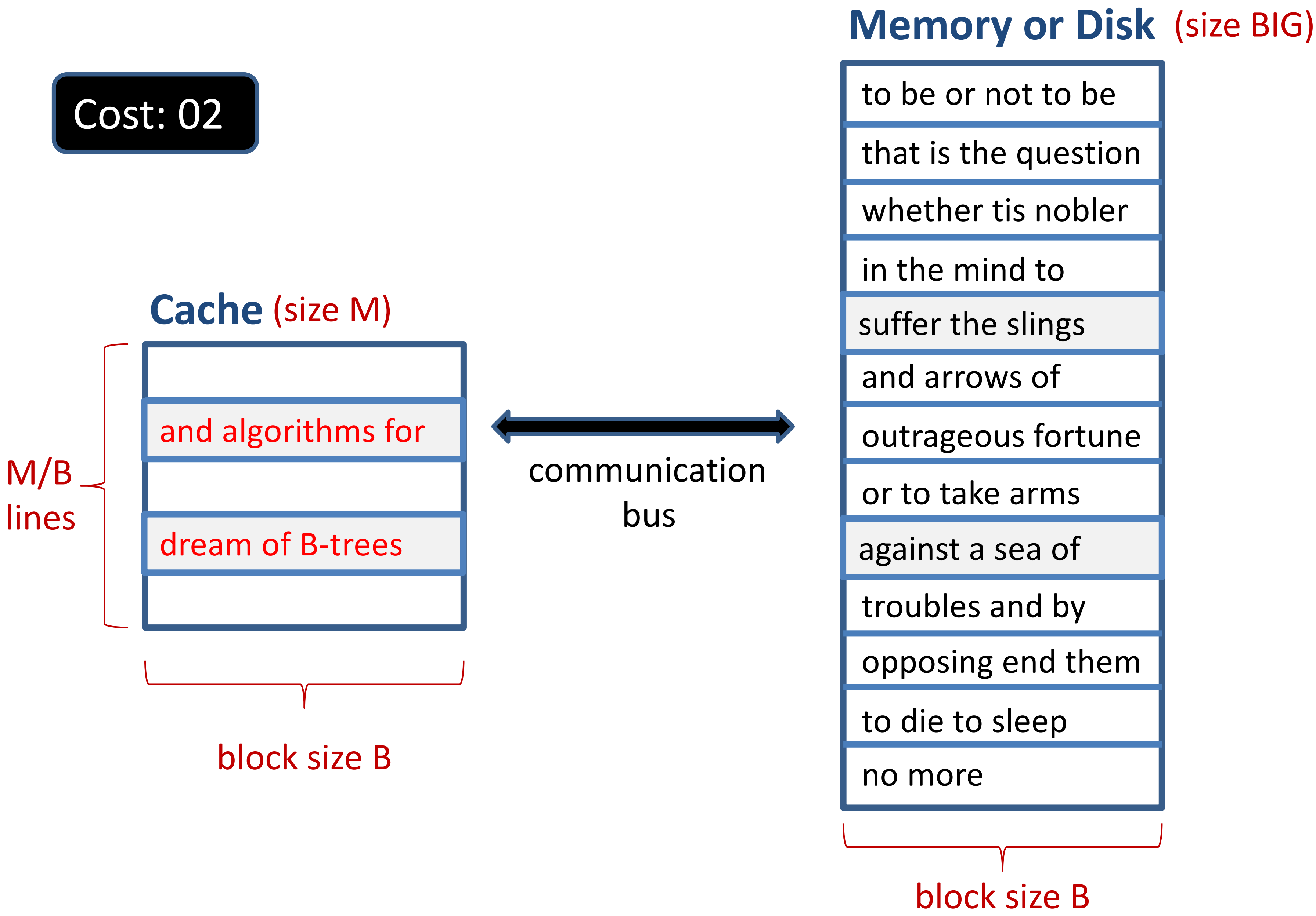
External Memory Model (Aggarwal, Vitter 1988)



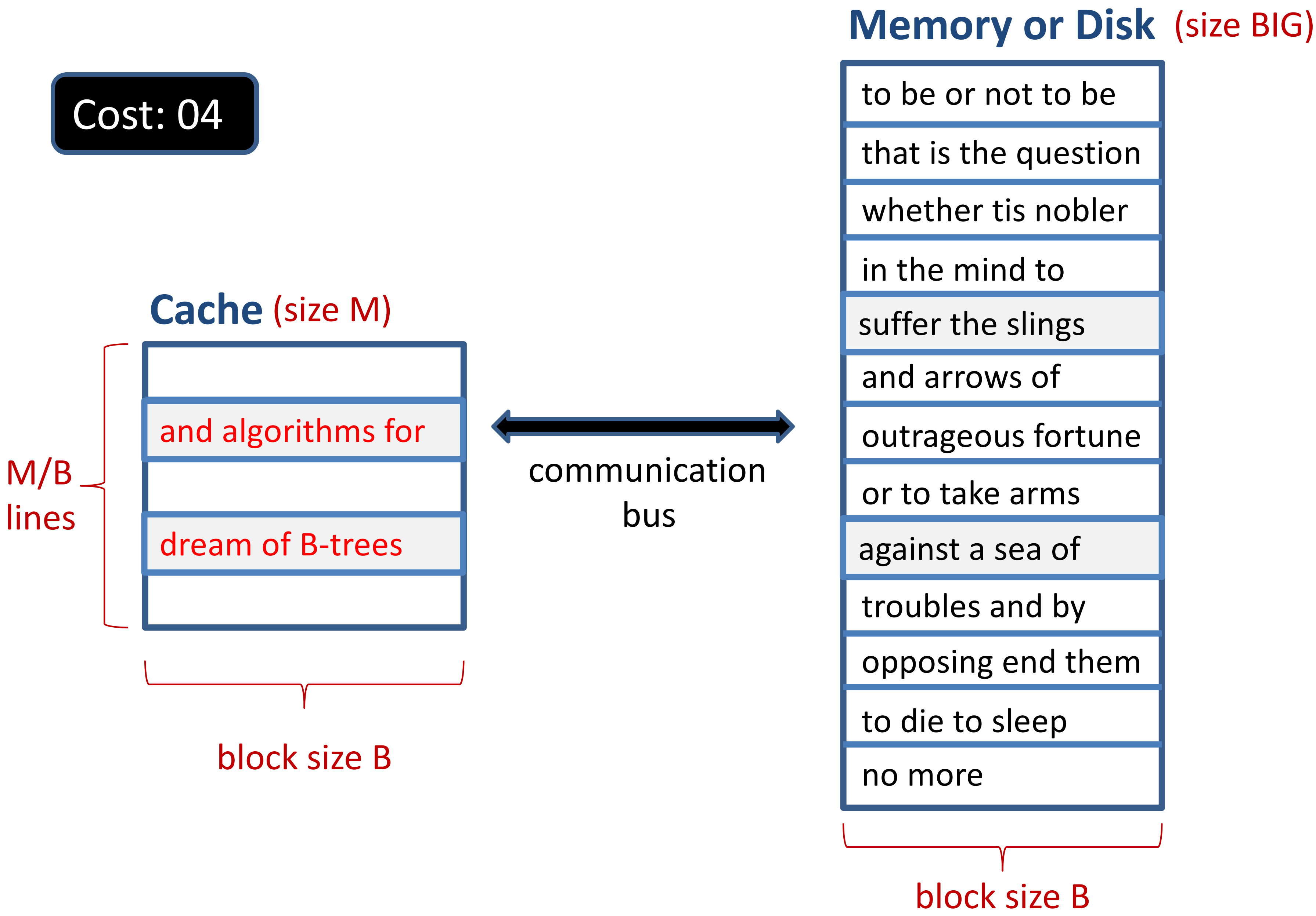
External Memory Model (Aggarwal, Vitter 1988)



External Memory Model (Aggarwal, Vitter 1988)



External Memory Model (Aggarwal, Vitter 1988)



External Memory Model (Aggarwal, Vitter 1988)

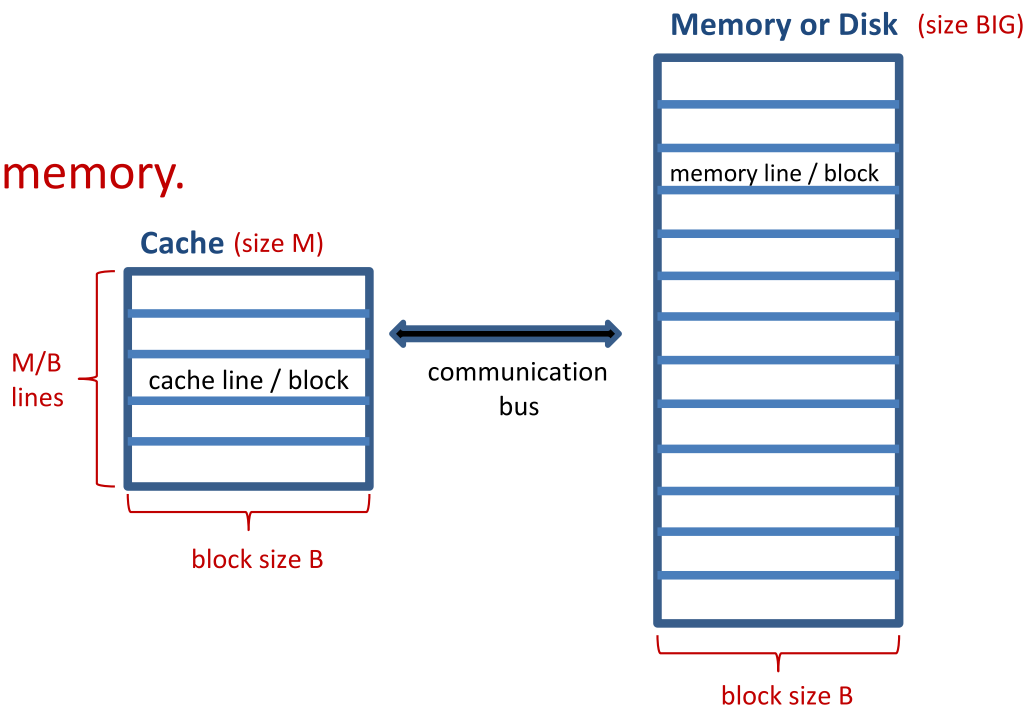
Rules:

On read / write operation:

1. Check if line is in cache. If so, perform operation in cache.
2. Else, expel a line from cache (write it back to memory).
3. Load requested line in cache.
4. Perform operation in cache.

Cost:

Number of lines read from or written to memory.



External Memory Model (Aggarwal, Vitter 1988)

Rules:

Which line to expel?

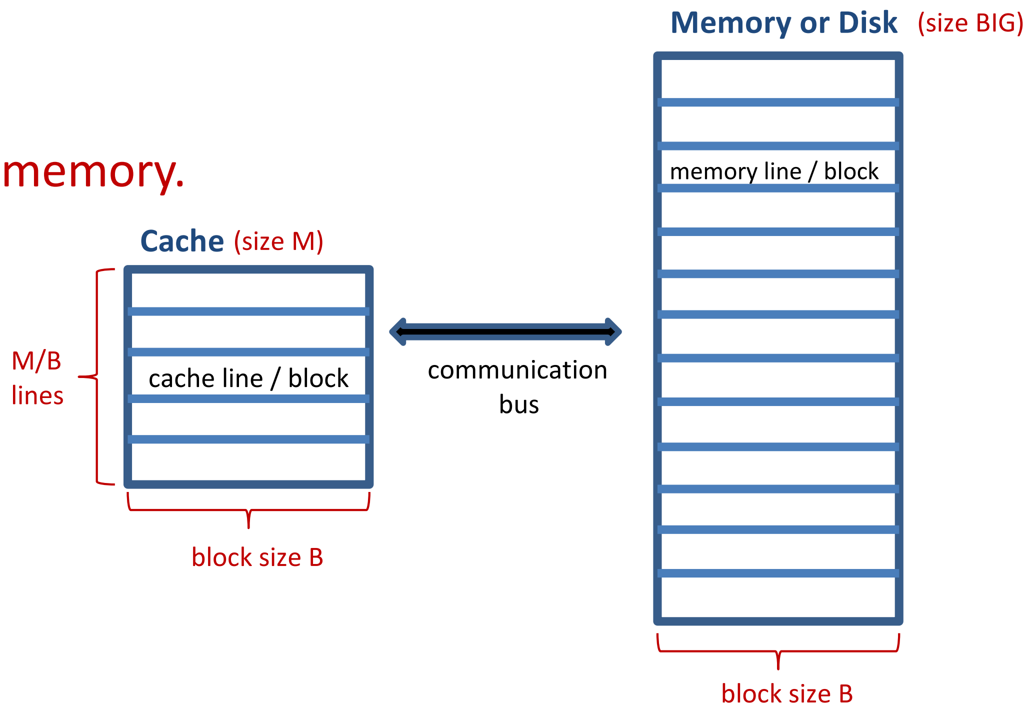
On read / write operation:

1. Check if line is in cache. If so, perform operation in cache.
2. Else, expel a line from cache (write it back to memory).
3. Load requested line in cache.
4. Perform operation in cache.

Where to store line in cache?

Cost:

Number of lines read from or written to memory.



External Memory Model (Aggarwal, Vitter 1988)

Rules:

On read / write operation:

1. Check if line is in cache. If so, perform operation in cache.
2. Else, expel a line from cache (write it back to memory).
3. Load requested line in cache.
4. Perform operation in cache.

Simplifications:

1. Only one level of cache. (Ignores L1, L2, etc.)
2. Only charges for memory access. (All other operations are free!)
3. Ideal caches. (Can store any line anywhere in the cache!)
4. Ideal replacement. (Ejects the line that will be not used for the longest time!)

External Memory Model (Aggarwal, Vitter 1988)

Rules:

On read / write operation:

1. Check if line is in cache. If so, perform oper
2. Else, expel a line from cache (write it back t
3. Load requested line in cache.
4. Perform operation in cache.

Three reasons:

- Works pretty well in practice.
- Simplifies analysis.
- One level usually dominates.

Simplifications:

1. Only one level of cache. (Ignores L1, L2, etc.)
2. Only charges for memory access. (All other operations are free!)
3. Ideal caches. (Can store any line anywhere in the cache!)
4. Ideal replacement. (Ejects the line that will be not used for the longest time!)

External Memory Model (Aggarwal, Vitter 1988)

Rules:

On read / write operation:

1. Check if line is in cache. If so, perform operation in cache.
2. Else, expel a line from cache (write it back to m
3. Load requested line in cache.
4. Perform operation in cache.

Good assumption?

- Usually, memory access dominates costs.
- Not true for compute-limited problems (e.g., TSP).

Simplifications:

1. Only one level of cache. (Ignores L1, L2, etc.)
2. Only charges for memory access. (All other operations are free!)
3. Ideal caches. (Can store any line anywhere in the cache!)
4. Ideal replacement. (Ejects the line that will be not used for the longest time!)

External Memory Model (Aggarwal, Vitter 1988)

Rules:

On read / write operation:

1. Check if … eration in cache.
2. Else, e … to memory).
3. Load …
4. Perfo…

Real caches?

- E.g., 8-way set associated
- Can simulate, lose only a constant factor (with resource augmentation).

Simplifications:

1. Only one level of cache. (Ignores L1, L2, etc.)
2. Only charges for memory access. (All other operations are free!)
3. Ideal caches. (Can store any line anywhere in the cache!)
4. Ideal replacement. (Ejects the line that will be not used for the longest time!)

External Memory Model (Aggarwal, Vitter 1988)

Rules:

On read / write oper

1. Check if line is in cache.
2. Else, expel a line from c
3. Load requested line in c
4. Perform operation in cache.

Replacement strategies?

- LRU: least recently used
- Ideal: farthest in the future
- Can simulate ideal with LRU, lose factof of 2 with resource augmentation.

Simplifications:

1. Only one level of cache. (Ignores L1, L2, etc.)
2. Only charges for memory access. (All other operations are free!)
3. Ideal caches. (Can store any line anywhere in the cache!)
4. Ideal replacement. (Ejects the line that will be not used for the longest time!)

For analysis: just let the algorithm decide!

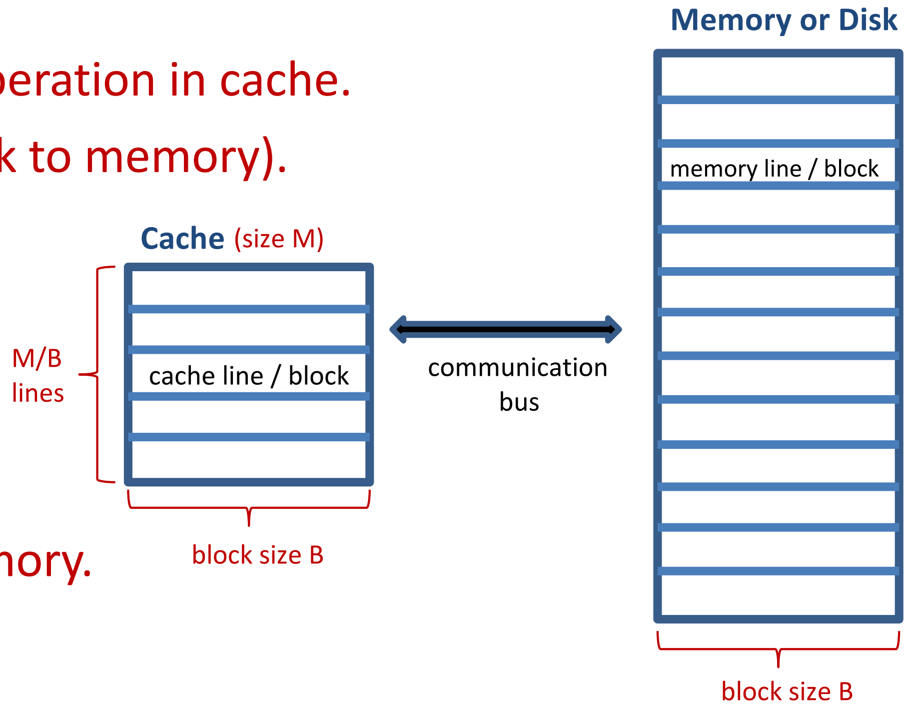
Cannot be better then optimal…

External Memory Model (Aggarwal, Vitter 1988)

Rules:

On read / write operation:

1. Check if line is in cache. If so, perform operation in cache.
2. Else, expel a line from cache (write it back to memory).
3. Load requested line in cache.
4. Perform operation in cache.



Cost:

Number of lines read from or written to memory.

Simplifications:

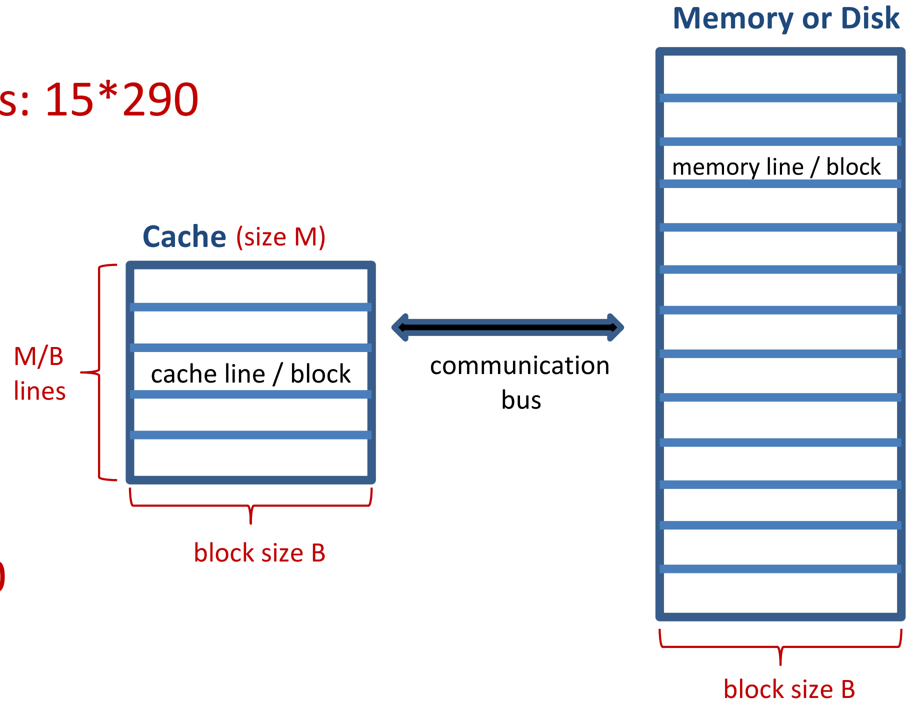
1. Only one level of cache. (Ignores L1, L2, etc.)
2. Only charges for memory access. (All other operations are free!)
3. Ideal caches. (Can store any line anywhere in the cache!)
4. Ideal replacement. (Ejects the line that will be not used for the longest time!)

External Memory Model (Aggarwal, Vitter 1988)

When is it useful?

Cache = L1/L2:

- Latency gap: 10 cycles vs. 300 cycles.
- Block size: 64 B.
- At best, every cache hit can save cycles: 15*290



Cache = Main Memory:

1. Latency gap: 300 cycles vs. 20,000,000
2. Block size: 16 KB
3. At best, every cache hit can save cycles: 16,000*20,000000

Predicting Performance

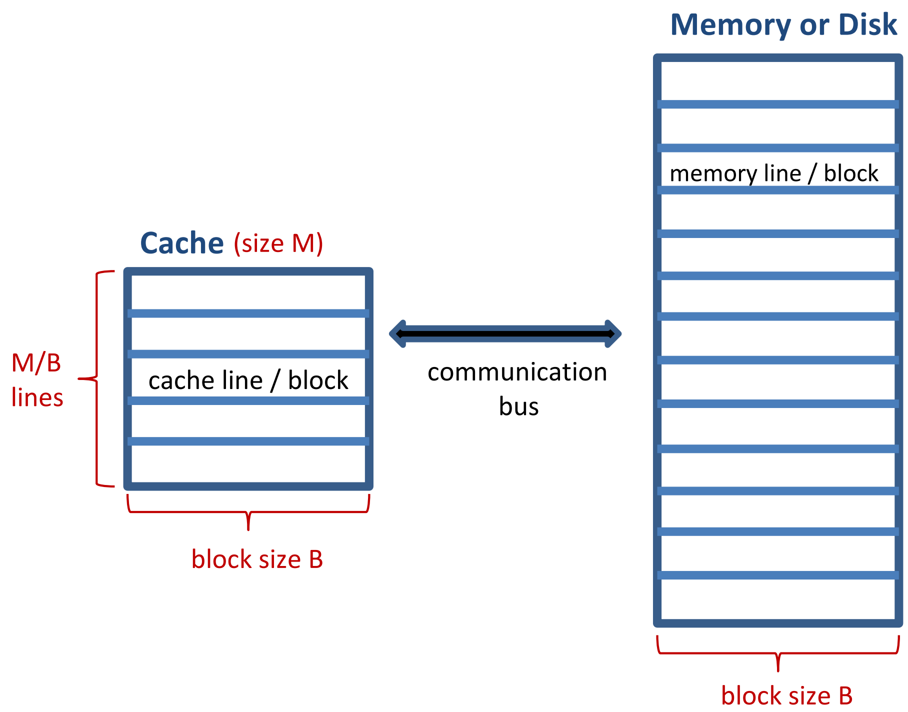
Example: Scanning data (size N)

1) Linked list

- ⇒ Classical analysis: O(N)
- ⇒ External memory: O(N)

2) Array

- ⇒ Classical analysis: O(N)
- ⇒ External memory: O(N/B)



Predicting Performance

Example: Searching data (size N)

1) Linked list

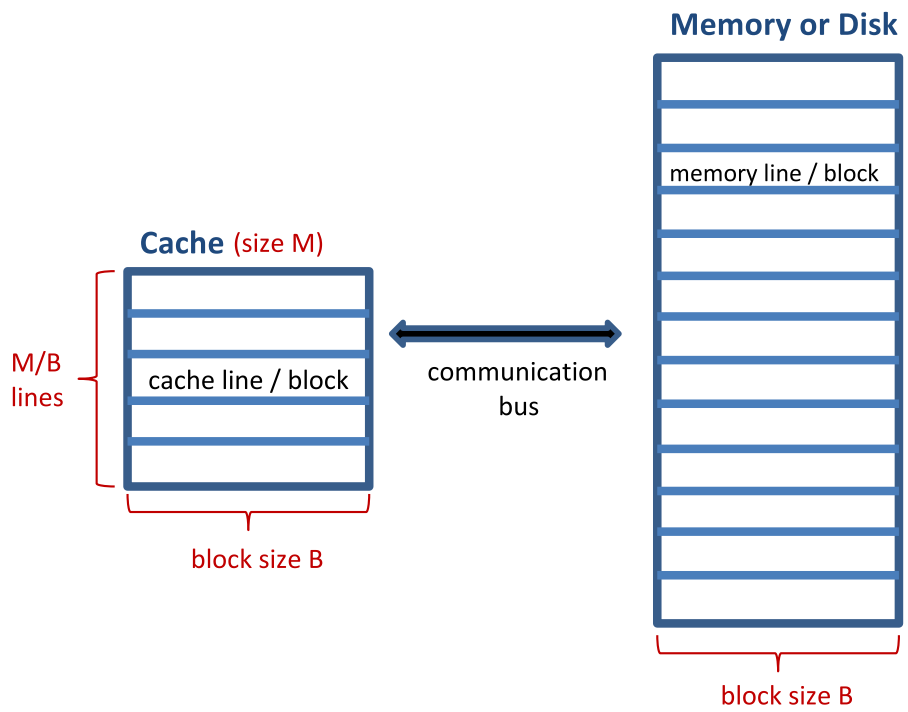
- ⇒ Classical analysis: O(N)
- ⇒ External memory: O(N)

2) Red-black tree

- ⇒ Classical analysis: O(log N)
- ⇒ External memory: O(log N)

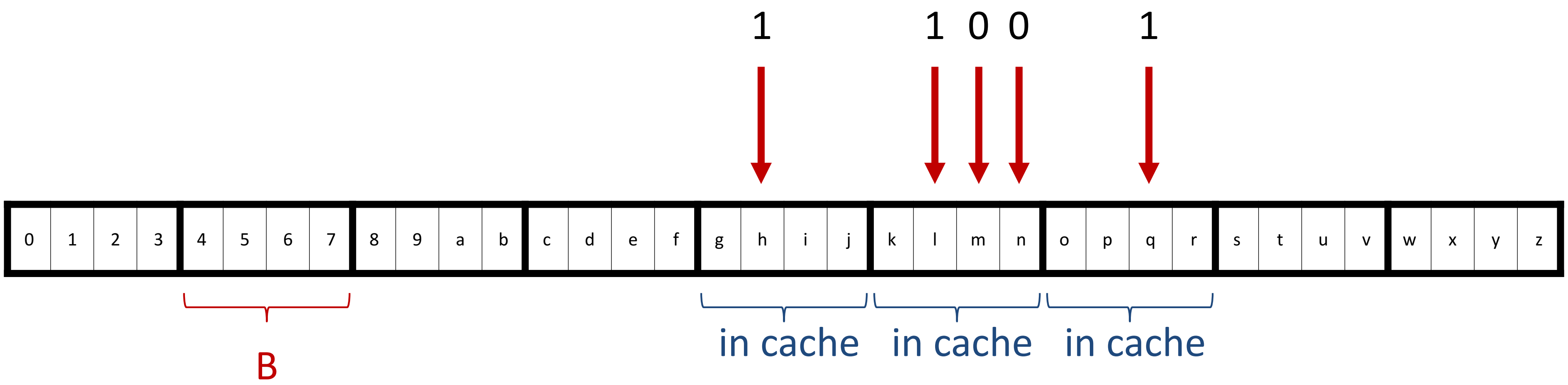
3) Array

- ⇒ Classical analysis: O(log N)
- ⇒ External memory: O(log (N/B))



Predicting Performance

Binary Search



binary-search(m)

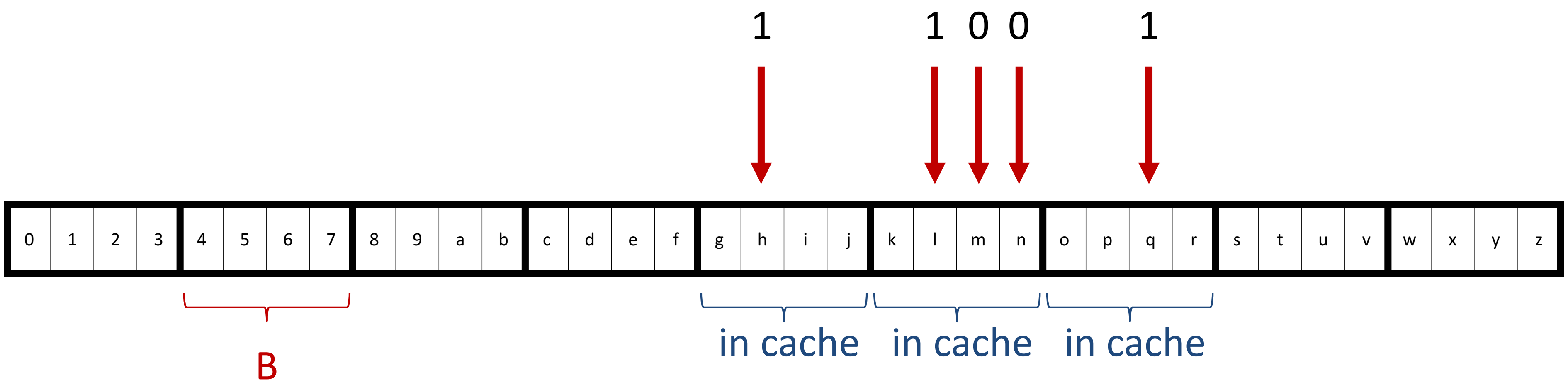
- query(h) ➔ cost = 1
- query(q) ➔ cost = 1
- query(l) ➔ cost = 1
- query(n) ➔ cost = 0
- query(m) ➔ cost = 0

Total cost: $O(\log(N/B))$

- N/B blocks total
- Binary search on N/B blocks.

Predicting Performance

Binary Search



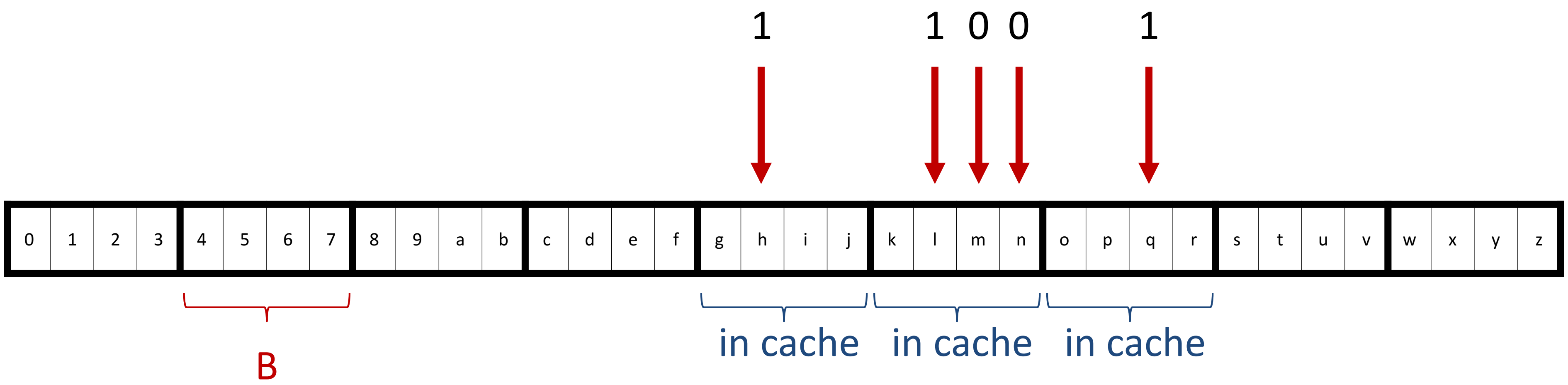
Comparison:

- Red-black tree: log(N)
- Array binary search: log(N/B) = log(N) – log(B)

Small improvement, if B is big!

Predicting Performance

Binary Search



Comparison:

Small improvement, if B is big!

- Red-black tree: log(N)
- Array binary search: log(N/B) = log(N) – log(B)
- B-tree: $\log_B(N)$ = log(N) / log(B)

Real improvement, even if B is small!

Predicting Performance

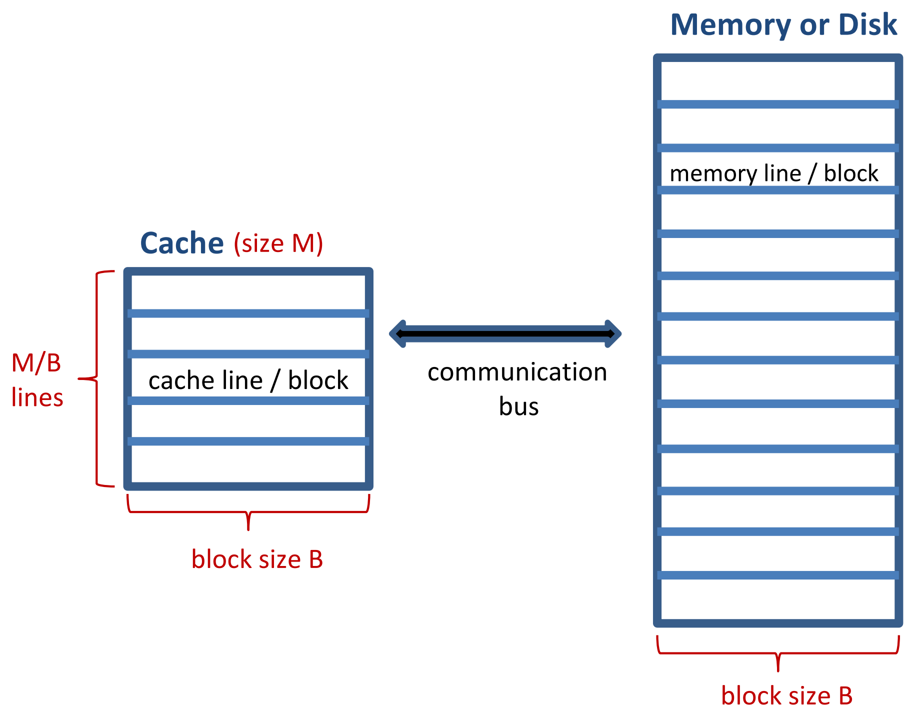
Example: Sorting data (size N)

1) QuickSort? MergeSort?

2) B-tree

⇒ Classical analysis: $O(N \log N)$

⇒ External memory: $O(N \log_B N)$



Predicting Performance

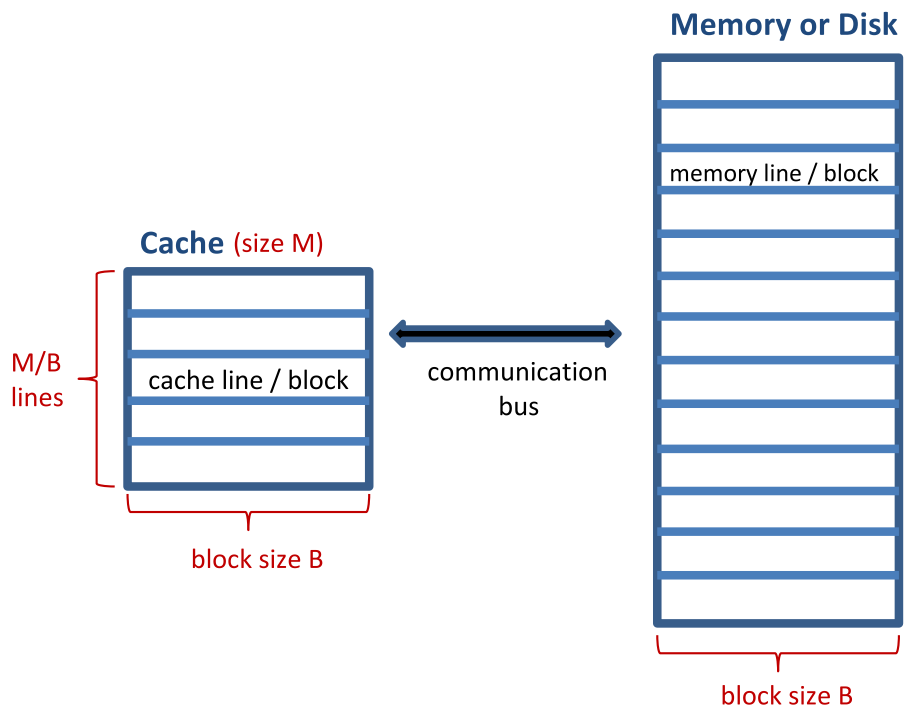
Example: Sorting data (size N)

1) QuickSort? MergeSort?

2) B-tree

⇒ Classical analysis: $O(N \log N)$

⇒ External memory: $O(N \log_B N)$



Predicting Performance

Example: Sorting data (size N)

1) QuickSort? MergeSort?

2) B-tree

⇒ Classical analysis: $O(N \log N)$

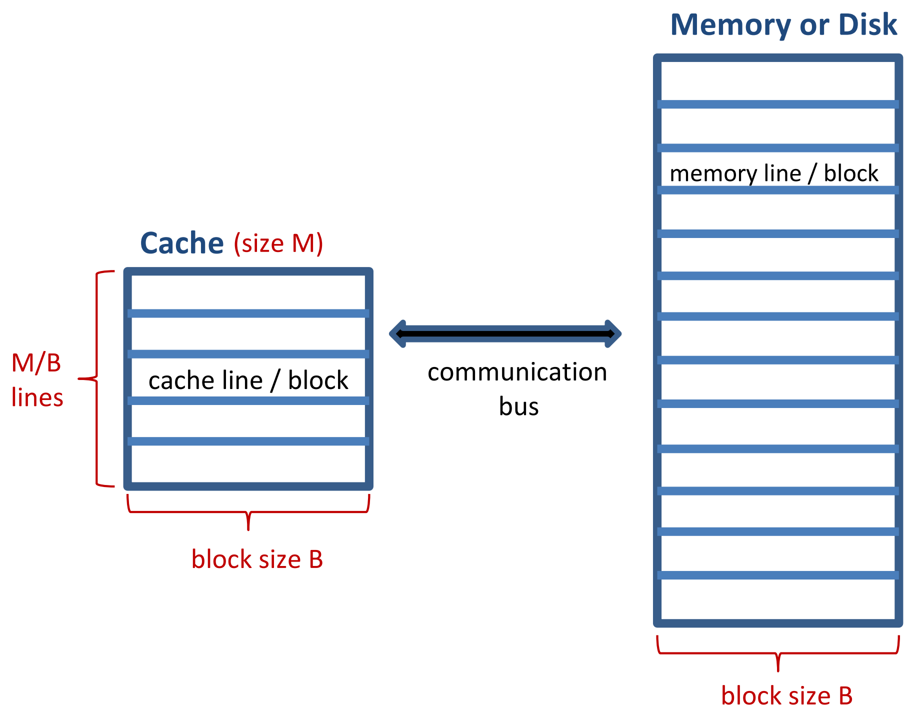
⇒ External memory: $O(N \log_B N)$

Optimal:

$$\mathrm{sort}(N) = O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$$

$$= O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{M}\right)$$

$$\log_{\frac{M}{B}}\frac{N}{B} = \log_{\frac{M}{B}}\left(\frac{N}{M}\frac{M}{B}\right) = \log_{\frac{M}{B}}\frac{N}{M} + 1$$



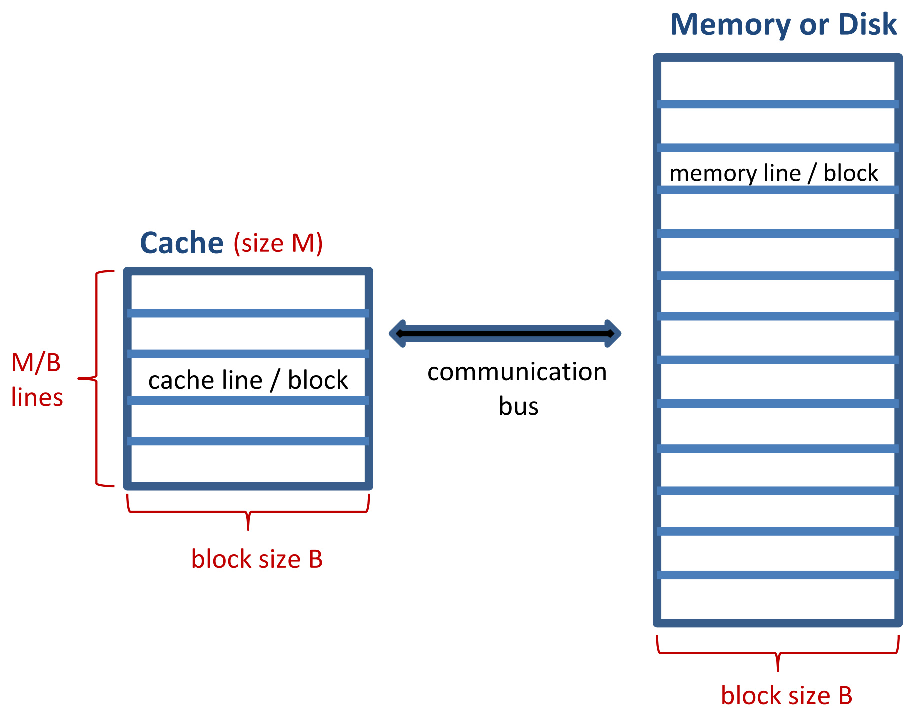
Predicting Performance

Example: Sorting data (size N)

Optimal:
$$\mathrm{sort}(N) = O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$$
$$= O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{M}\right)$$

Notes:

- Size of cache (M) matters.
- 3 standard solutions
  - External MergeSort
  - External QuickSort
  - BufferTree Sort
- One "cache oblivious" solution
  - FunnelSort



Predicting Performance

Example: Graphs (V nodes, E edges)

1) Priority Queue: $O\left(\frac{1}{B}\log_{M/B}\frac{V}{B}\right)$

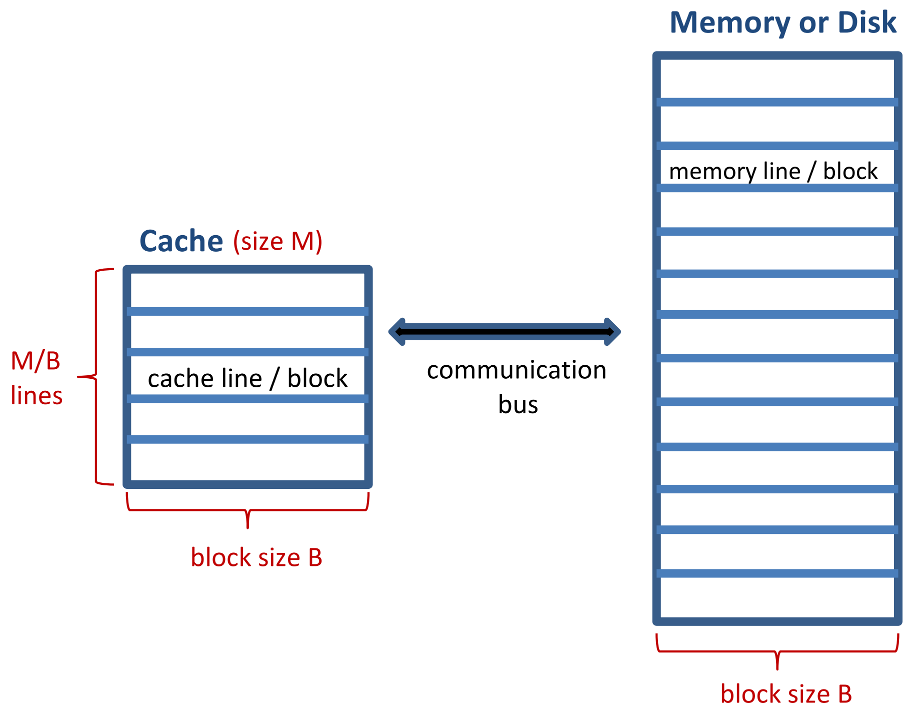
2) Unweighted shortest paths: $O\left(V + \frac{E}{B}\log_{M/B}\frac{E}{B}\right)$

3) Dijkstra's: $O\left(V + \frac{E}{B}\log\frac{E}{M}\right)$

4) Unweighted APSP:

$$O\left(\frac{VE}{B}\log_{\frac{M}{B}}\frac{E}{B}\right)$$

For < 65PB, 4000x faster than O(VE).



Today's Plan

Searching and Sorting

1. B-trees
   - ⇒ Algorithm
   - ⇒ Amortized analysis

2. Buffer trees
   - ⇒ Write-optimized data structures
   - ⇒ Buffered data structures
   - ⇒ Amortized analysis

3. van Emde Boas Search Tree
   - ⇒ Cache-oblivious algorithms
   - ⇒ van Emde Boas memory layout

B-trees

Basic facts

- One of the most important data structures out there today. (Variants used in all major databases.)
- Very fast. (Not just asymptotic analysis, but in practice nearly impossible to beat a well-implemented B-tree.)
- Benefit comes both from good cache performance, low overhead, good parallelization, etc.

B-trees

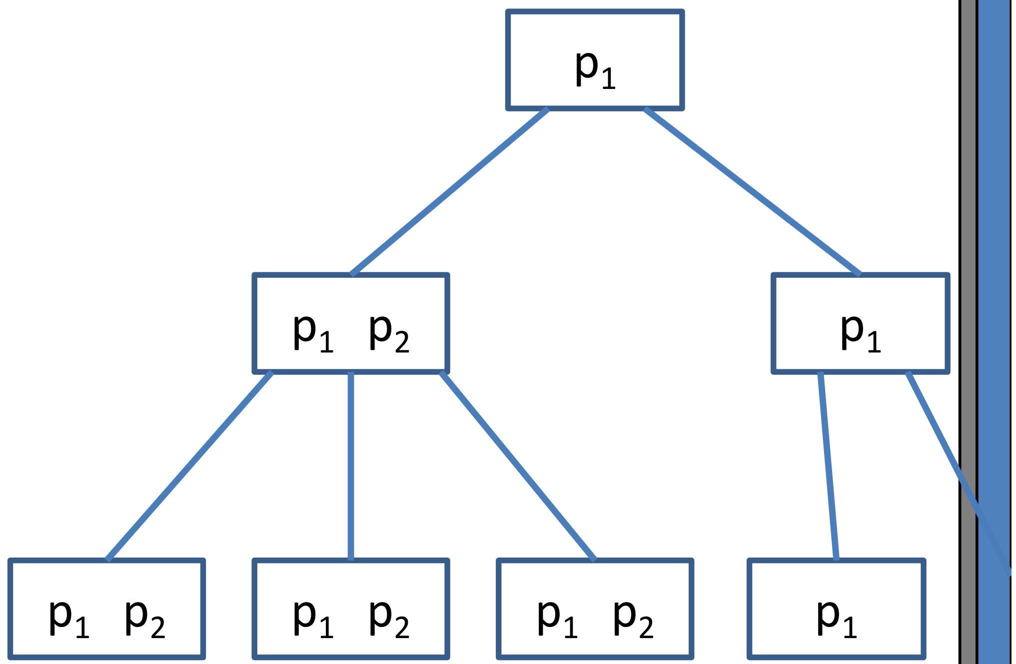
Basic facts

- One of the most important data structures out there today. (Variants used in all major databases.)
- Very fast. (Not just asymptotic analysis, but in practice nearly impossible to beat a well-implemented B-tree.)
- Benefit comes both from good cache performance, low overhead, good parallelization, etc.

(a, b)-trees

Basics:

- Tree structure.
- Satisfies search property.
- $b \geq 2a$
  (e.g., $a = B$, $b = 2B$)
- All keys stored in leaves
- Internal nodes store *pivots* to guide search.



(a, b)-trees

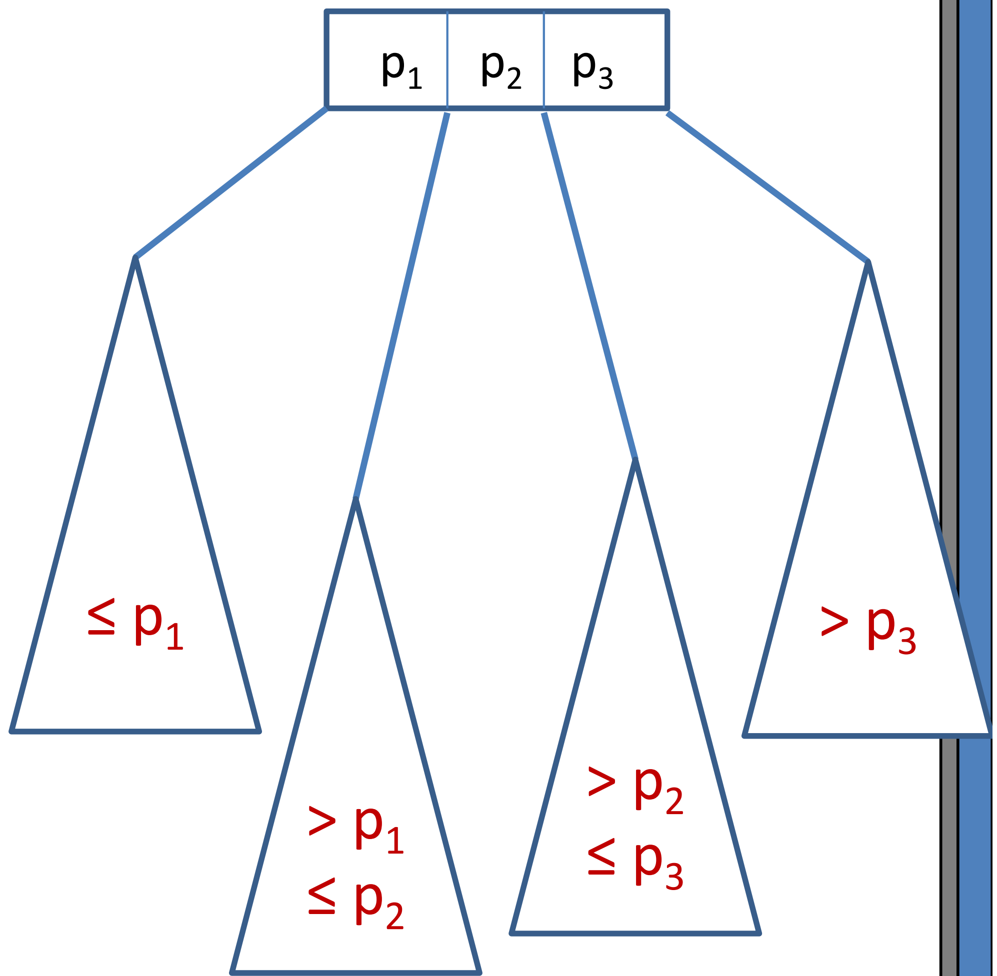
Each node stores:

- parent
- set of *pivots* $p_1, p_2, \ldots$
- pointers to *sub-trees* $T_1, T_2, \ldots$

Search property:

For subtree T_j:

all the keys in T_j have values in the range $(p_{j-1}, p_j]$



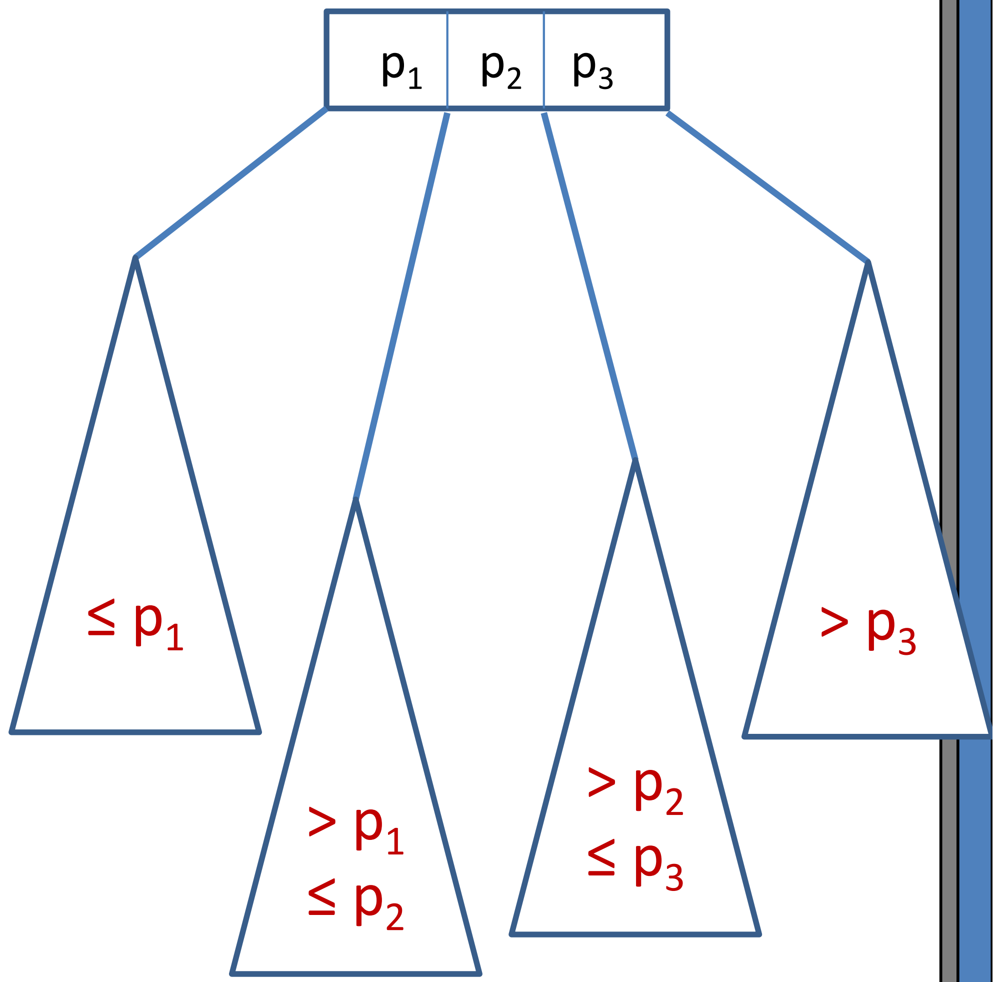
(a, b)-trees

Each node stores:

- parent
- set of *pivots* p_1, p_2, …
- pointers to *sub-trees* T_1, T_2, …

Question:

How should a node store its keys and sub-tree pointers?



(a, b)-trees

Each node stores:

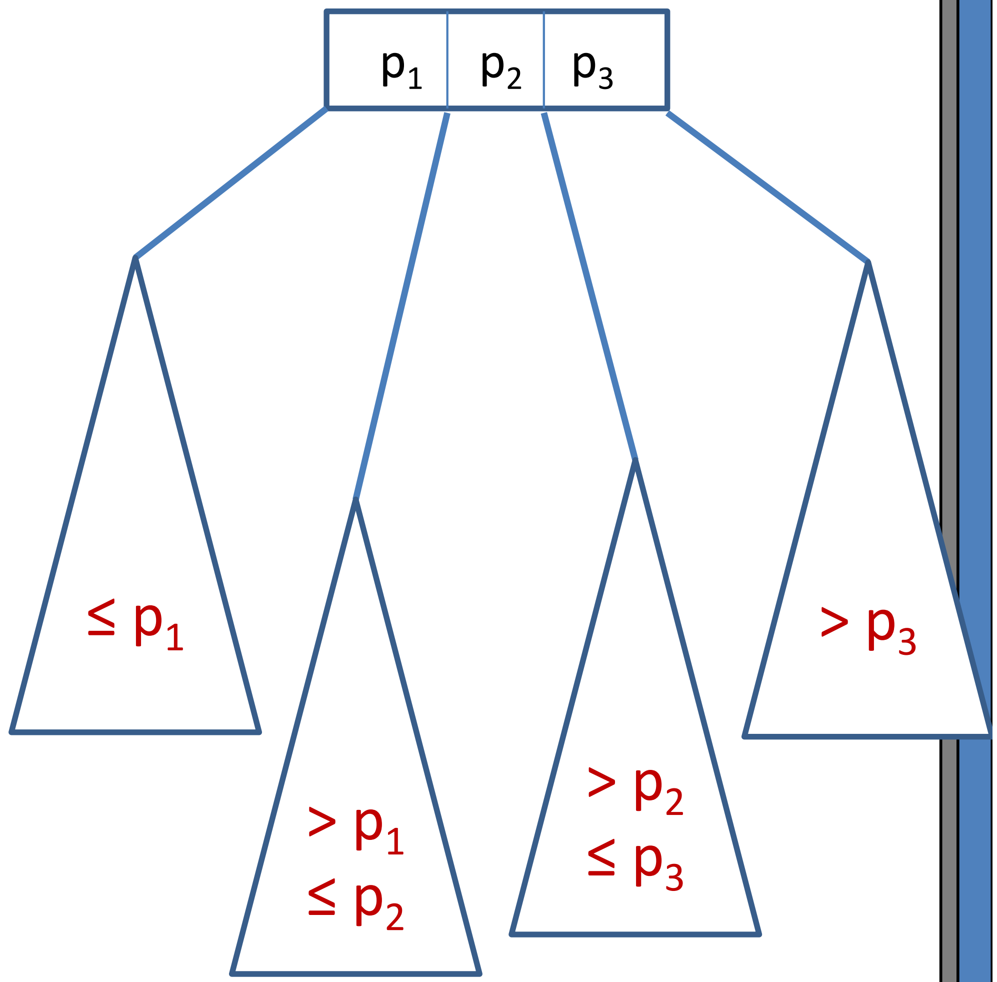
- parent
- set of *pivots* $p_1, p_2, \ldots$
- pointers to *sub-trees* $T_1, T_2, \ldots$

Question:

How should a node store its keys and sub-tree pointers?

Possible answers:

1. In this model, does not matter.
2. In practice, use a small tree.
3. Can use a recursive B-tree, optimized for a different level cache!



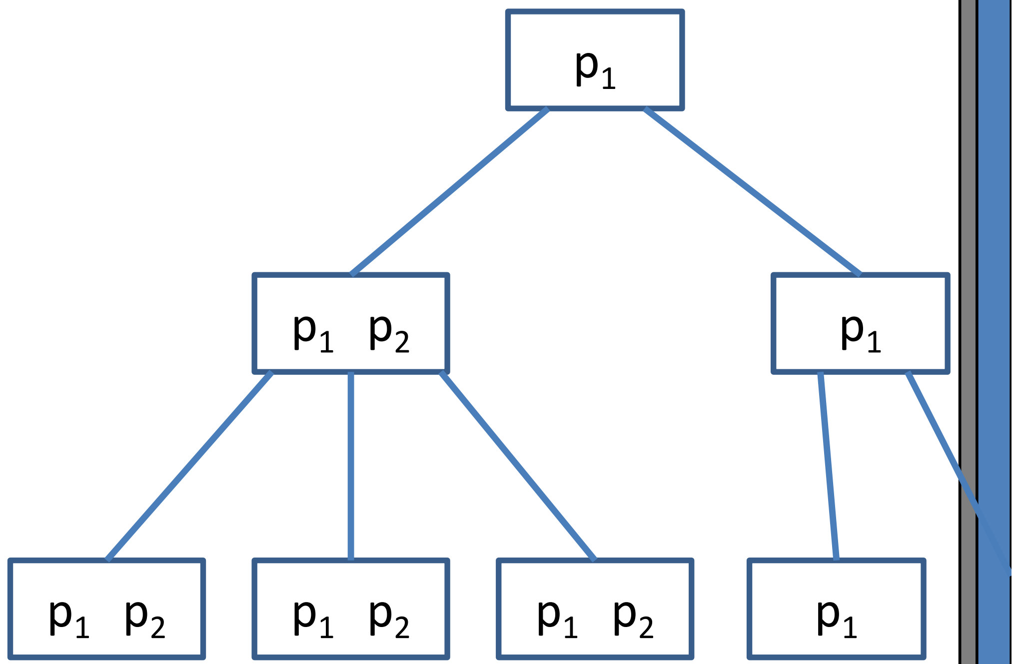
(a, b)-trees

Basics:

- tree structure
- satisfies search property
- $b \geq 2a$
  (e.g., $a = B$, $b = 2B$)

Rules:

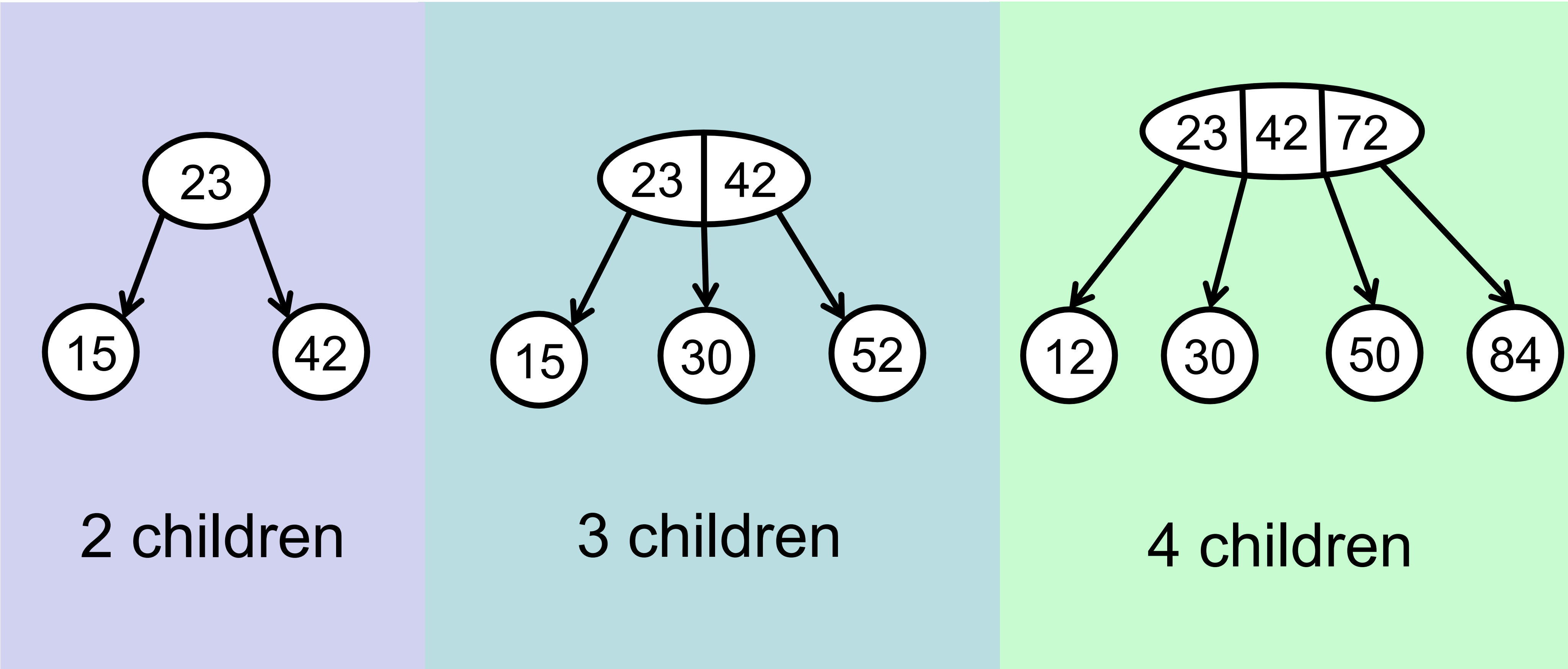
1. Root has ≥ 2 children.
2. Non-root nodes have $\geq a$ children.
3. All nodes have $\leq b$ children.
4. All leaves have the same depth.
5. For all leaves: $a \leq \#keys \leq b$



Ex: (2,4) Trees

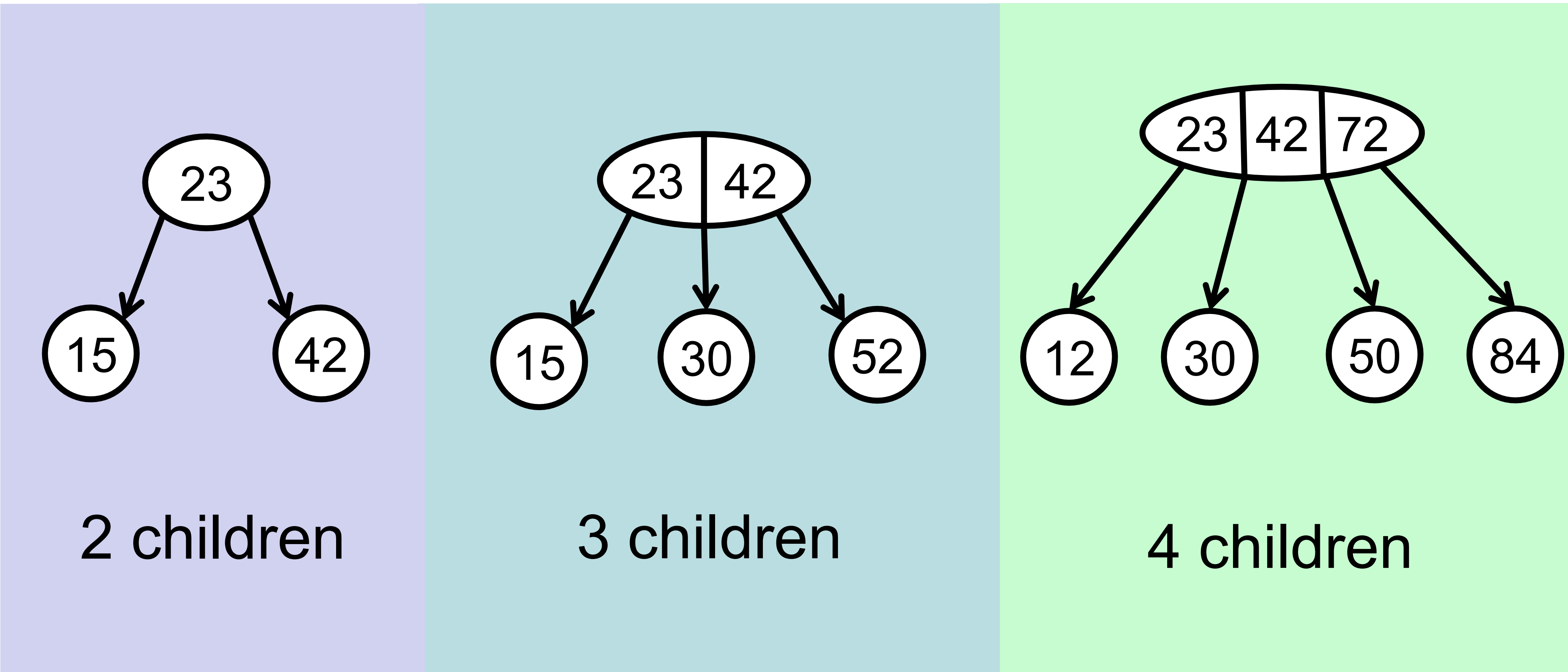
Rules #1--3:

- Every non-leaf node has either:
  2 or 3 or 4 children



Ex: (2,4) Trees

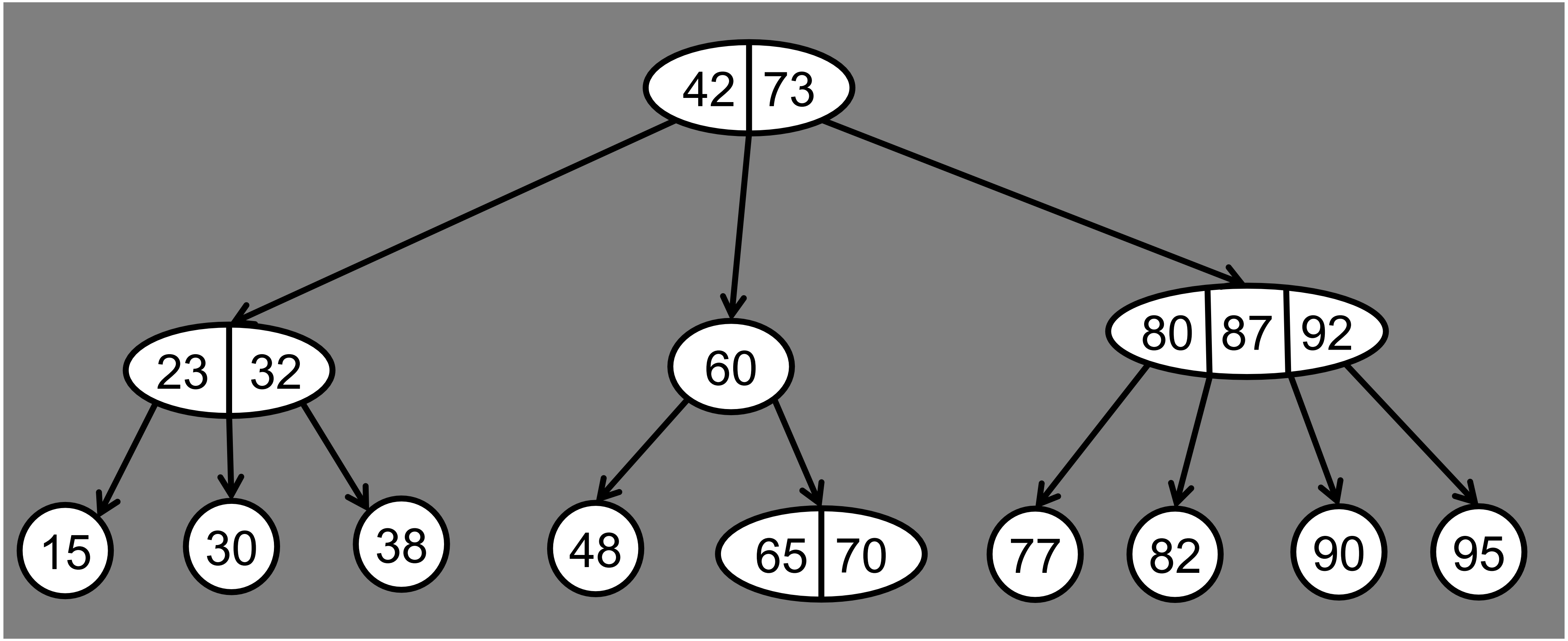
Search property:



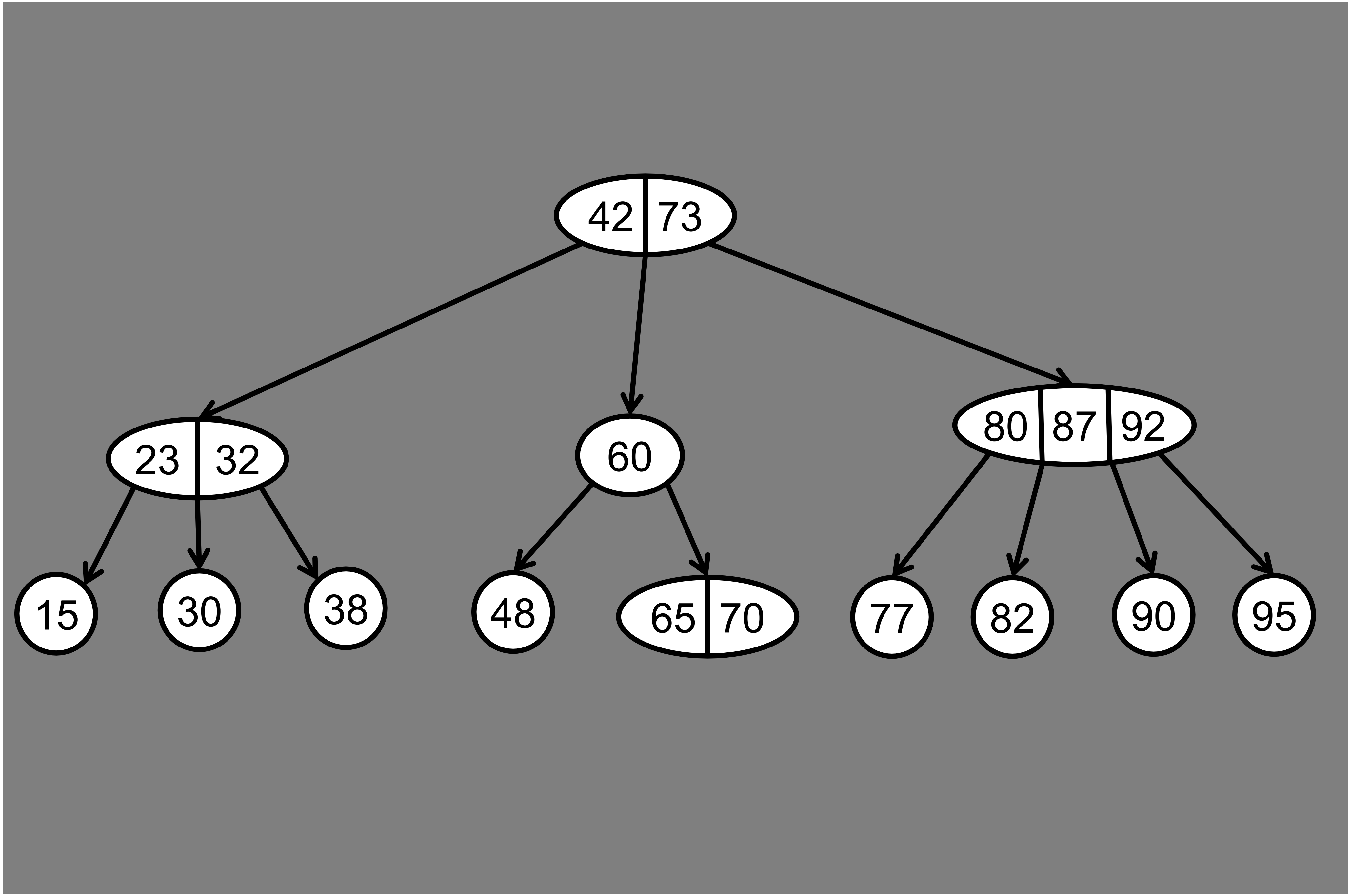
Ex: (2,4) Trees

Rule 4: Every leaf has the same depth.

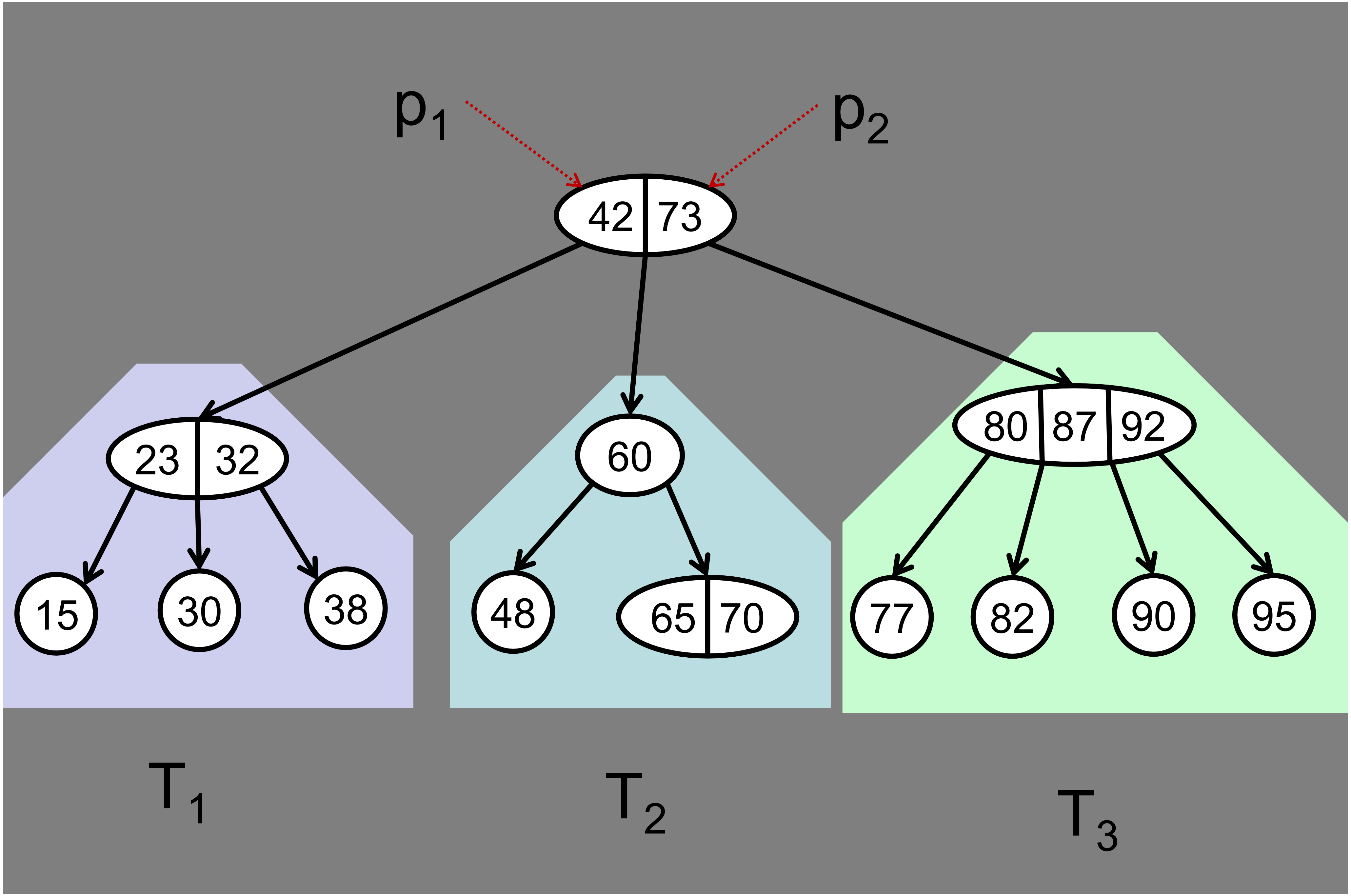
– Every path from root->leaf is the same length.



Ex: (2,4) Trees

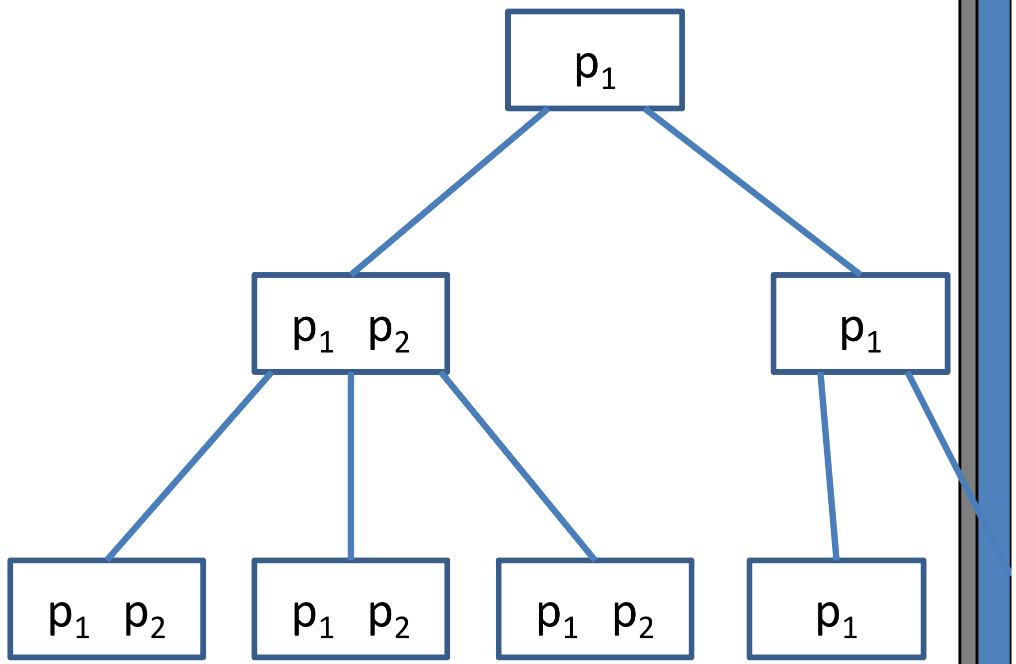


Ex: (2,4) Trees

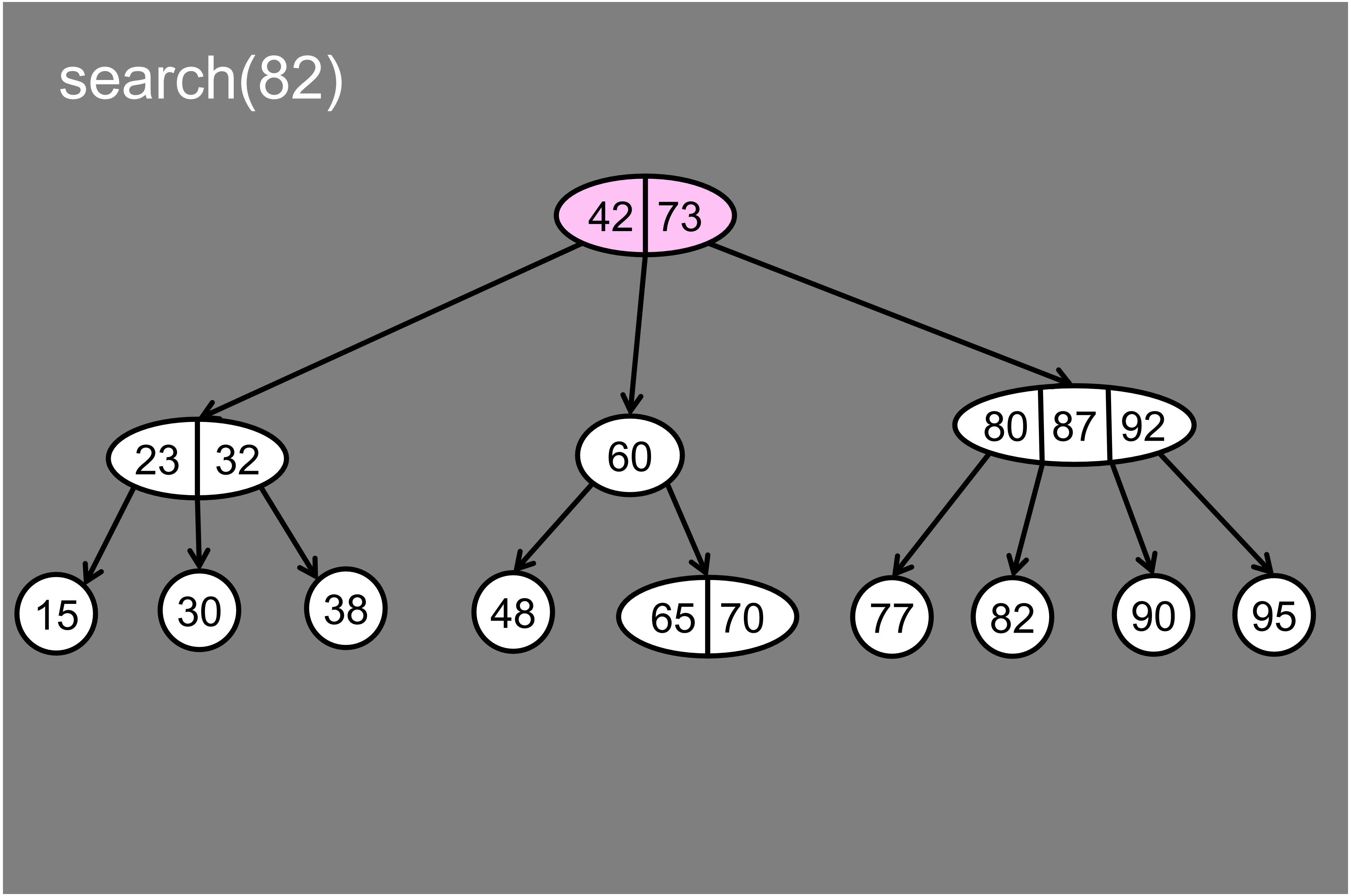


(a, b)-trees

search(k):
- v = root
- while not leaf(v):
  - if $k \le p_1$ then $v = T_1$
  - else let $c = \max(j : k > p_j)$
    - $v = T_{c+1}$

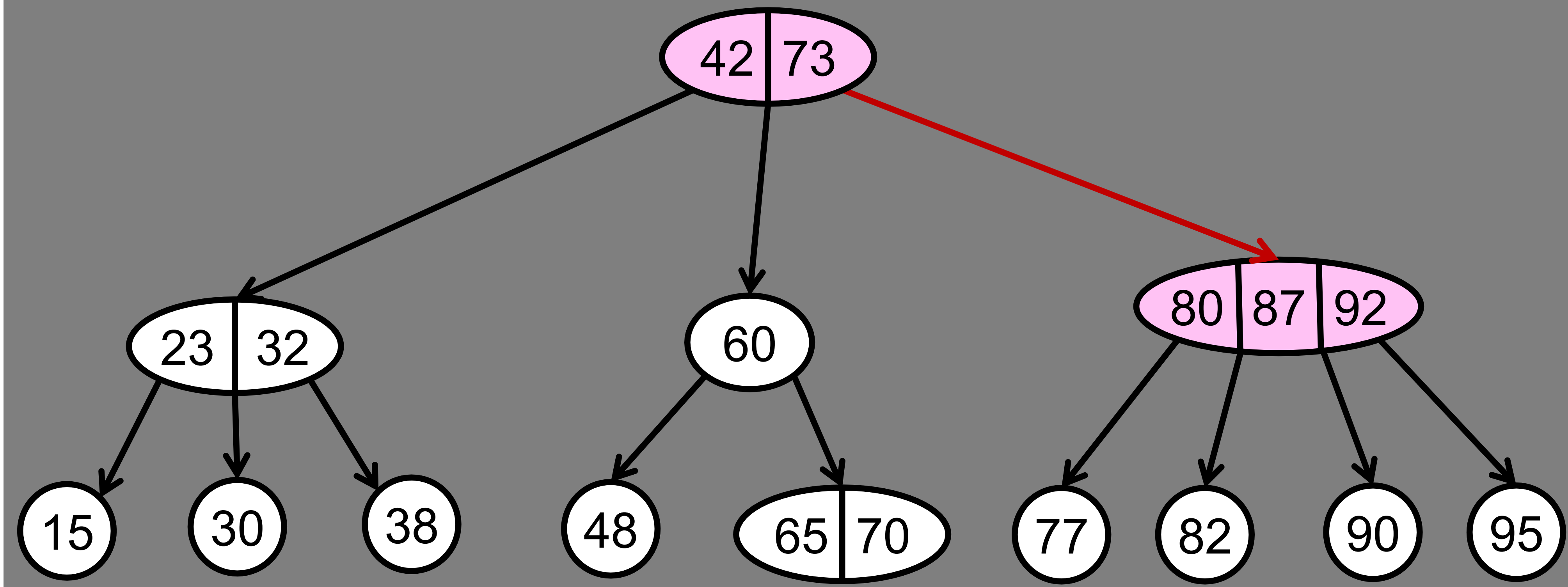


Ex: (2,4) Trees

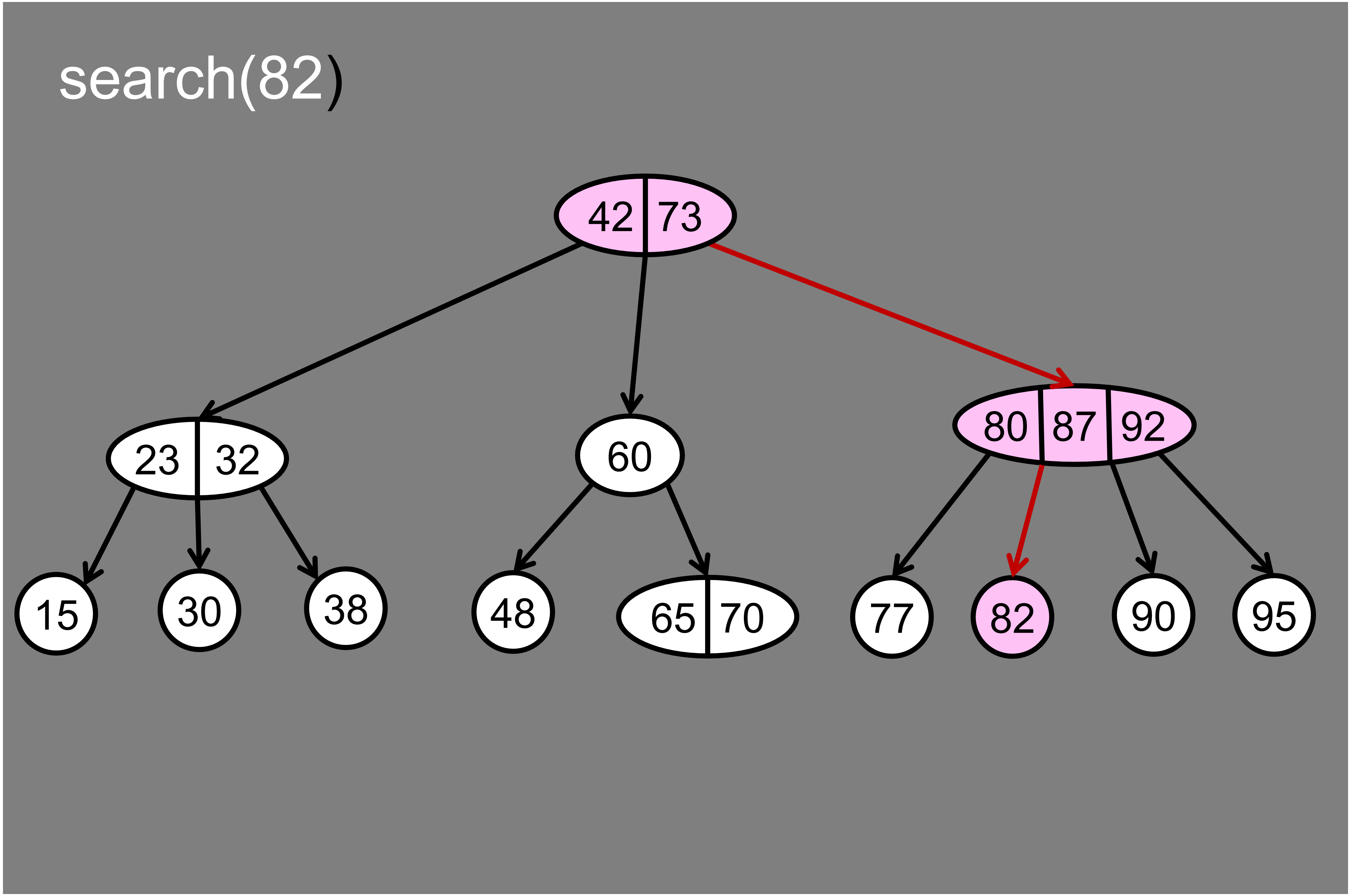


Ex: (2,4) Trees

search(82)

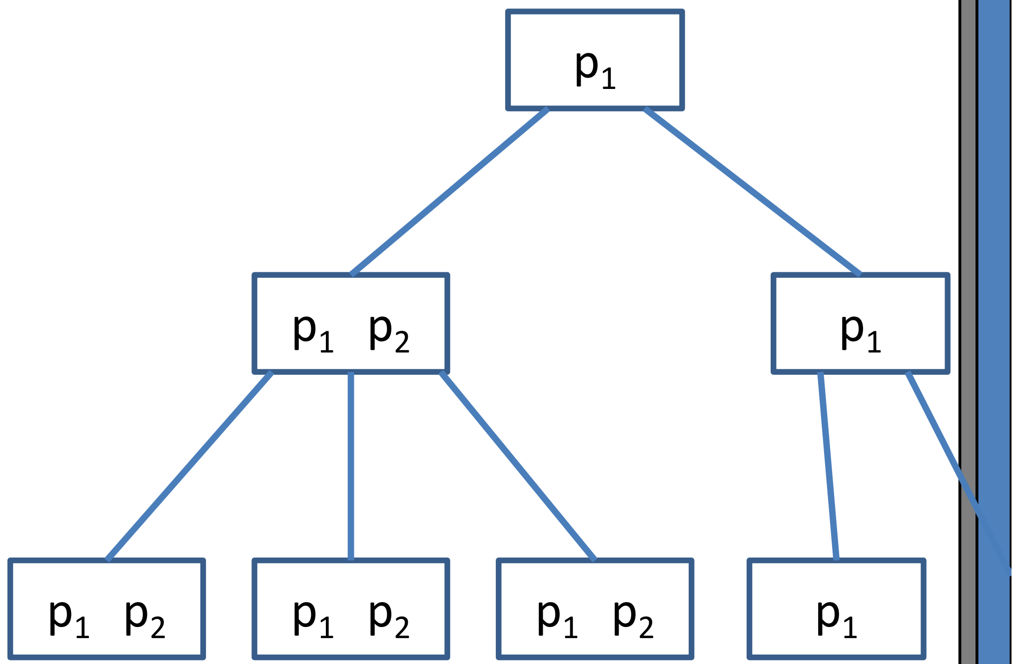


Ex: (2,4) Trees



(a, b)-trees

Claim:
Search works.

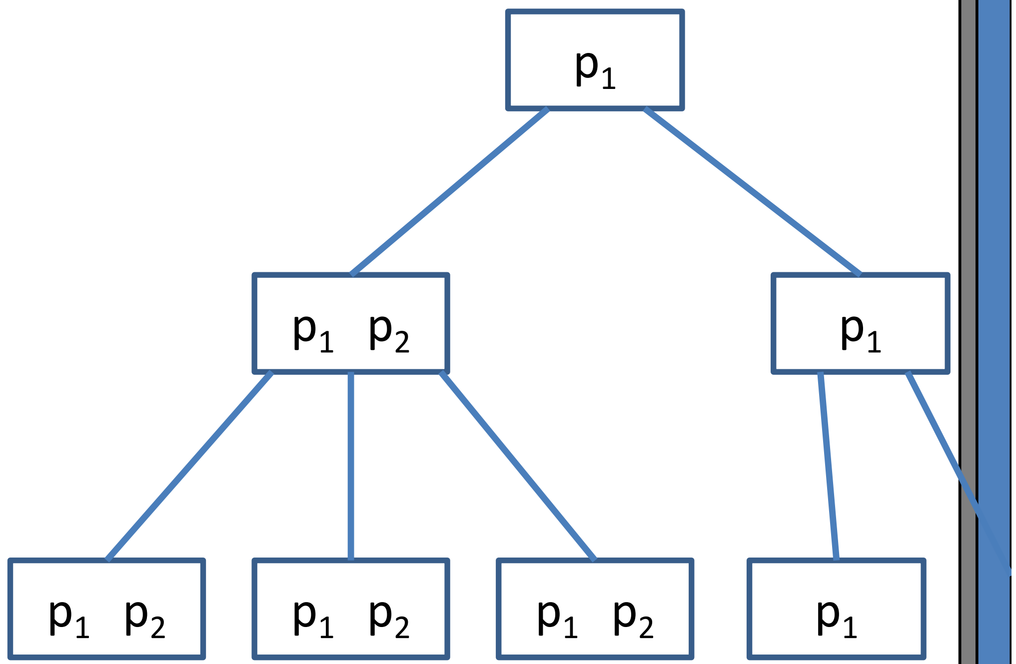


(a, b)-trees

Claim:

An (a,b)-tree with n keys

has height: $\leq \log_a \left(\frac{n}{a}\right) + 1$



(a, b)-trees

Claim:

An (a,b)-tree with n keys

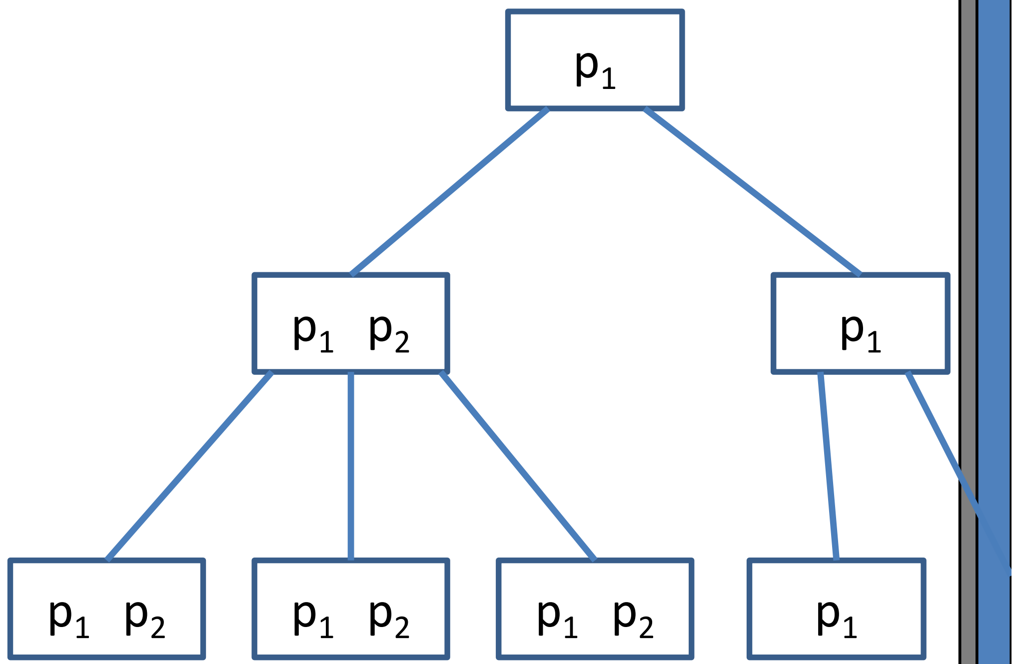
has height: $\leq \log_a \left(\frac{n}{a}\right) + 1$

Proof:

- At most (n/a) leaves.
- Every node except the root has degree at least a.
- So a node at height $\log_a \left(\frac{n}{a}\right)$ has at least:

  $$\geq a^{\log_a \left(\frac{n}{a}\right)} \geq \frac{n}{a}$$

  leaves.
- So the children of the root have maximum height: $\log_a \left(\frac{n}{a}\right)$



(a, b)-trees

Claim:

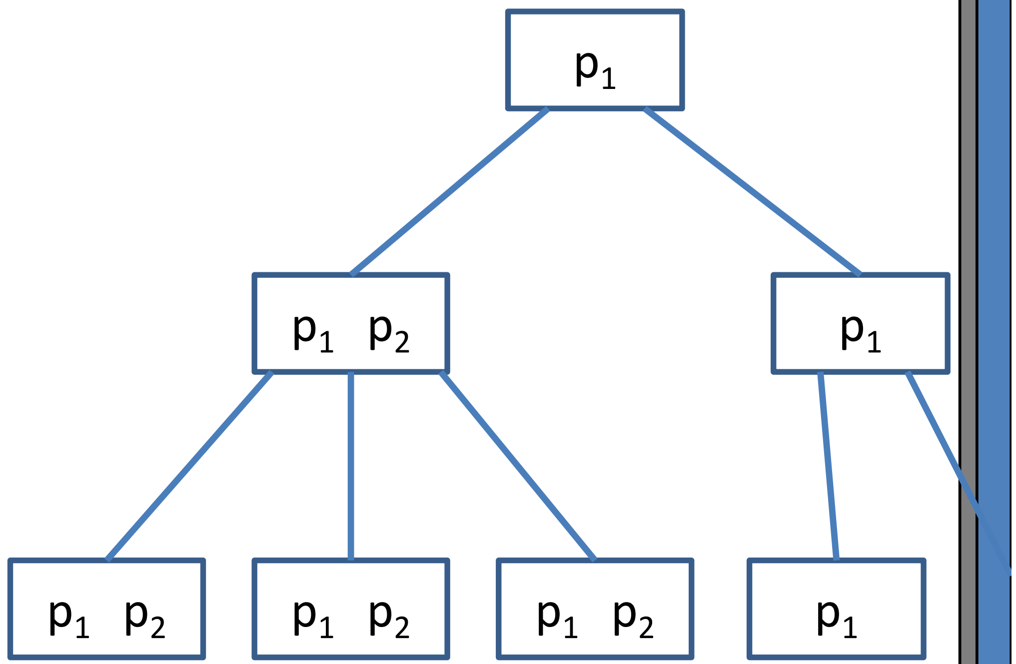
An (a,b)-tree with n keys

has height: $\leq \log_a \left(\frac{n}{a}\right) + 1$

Corollary:

If a≥B, then an (a,b)-tree with

n keys has height: $O(\log_B n)$

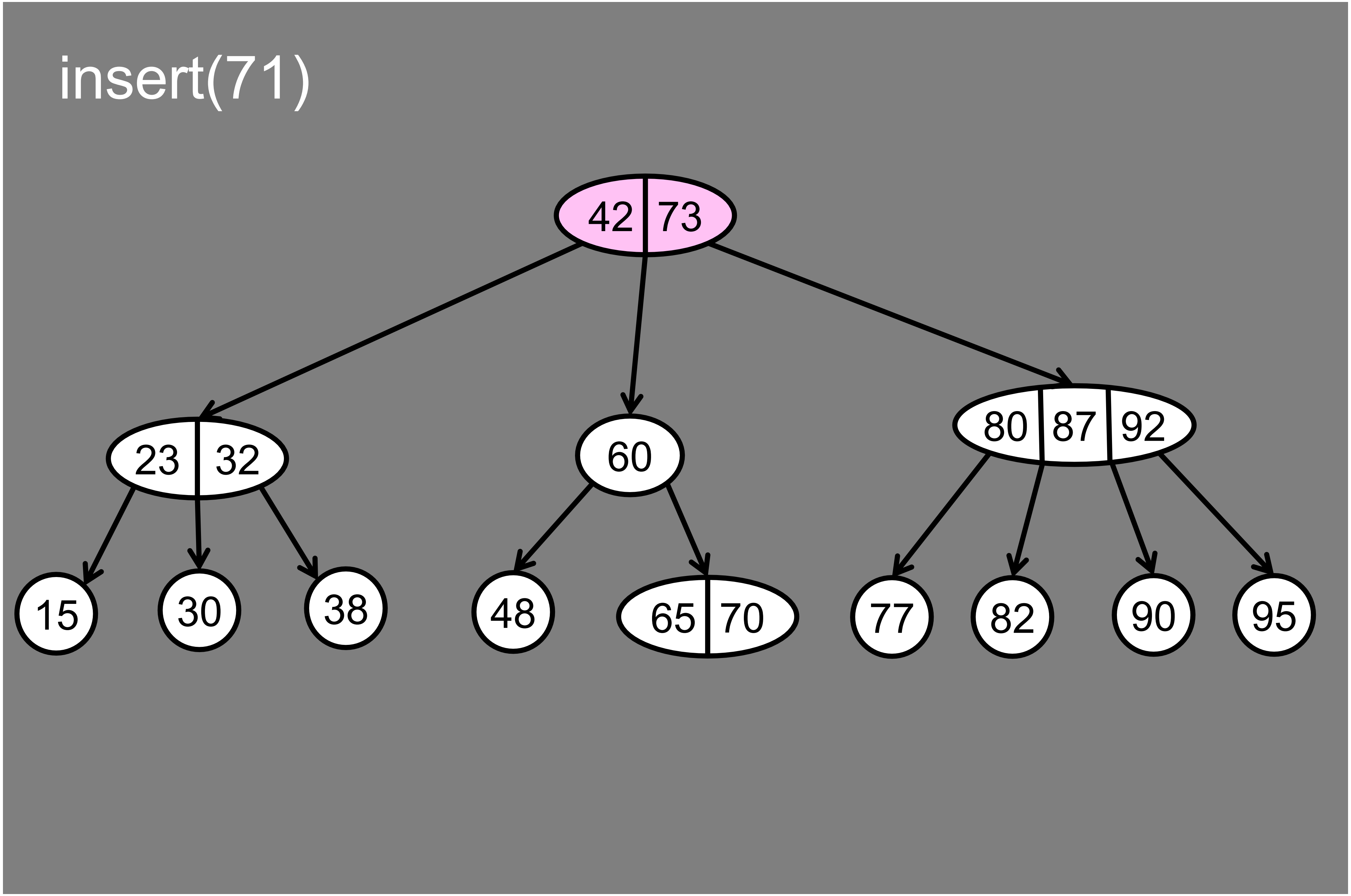


(a, b)-trees

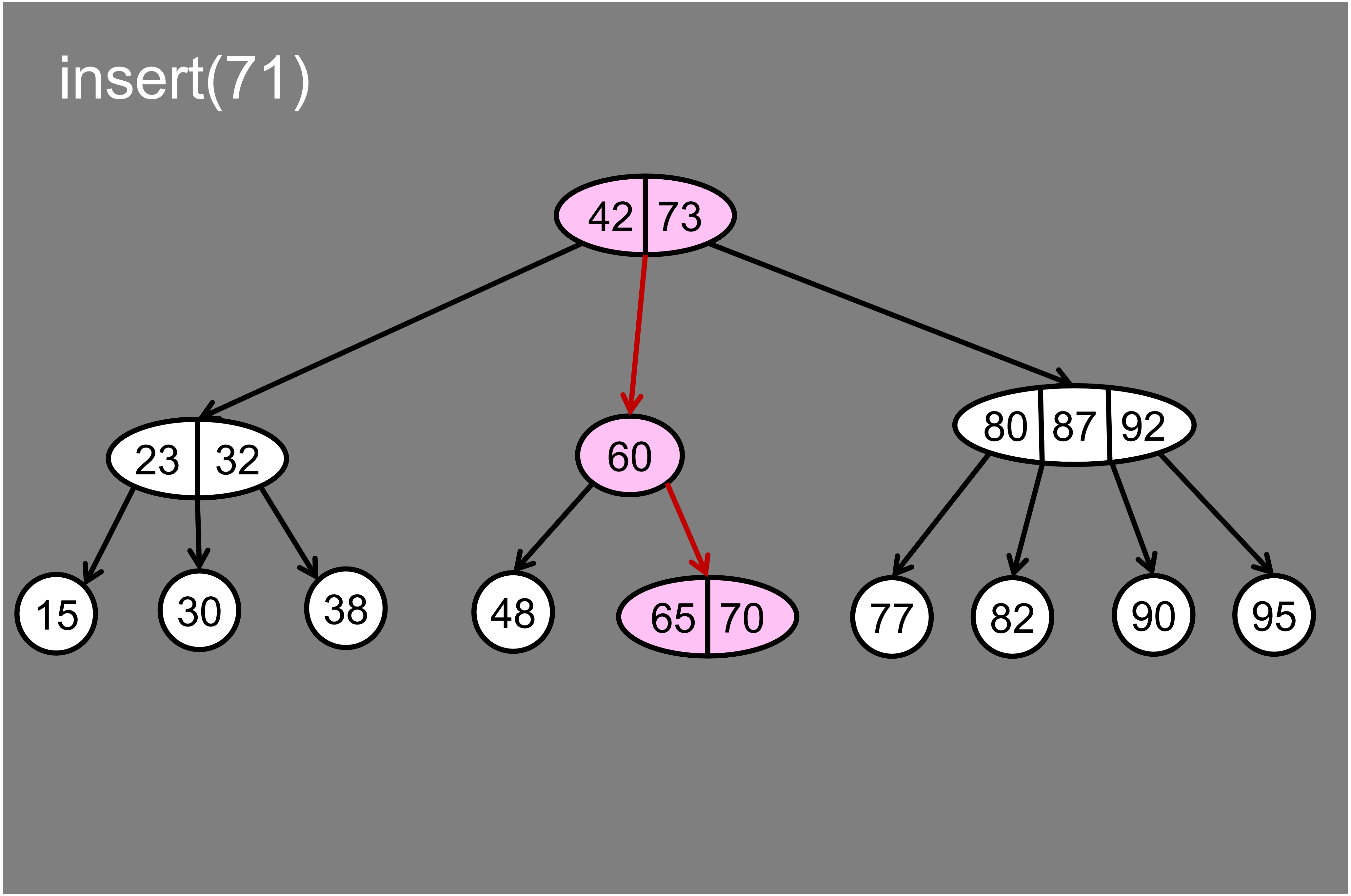
insert(k):

1. Search for leaf node v containing key k
2. Add key k to leaf node v.
3. If node v has > b keys:
   - Split node v into two.
     Each piece has $> b/2 \geq a$ keys.
   - Call new nodes x and y.
   - k = max element in x. (If v is not a leaf, remove k from node v.)
   - Recursively insert k into parent(v).
   - Update parent/child pointers of x, y, parent(v).

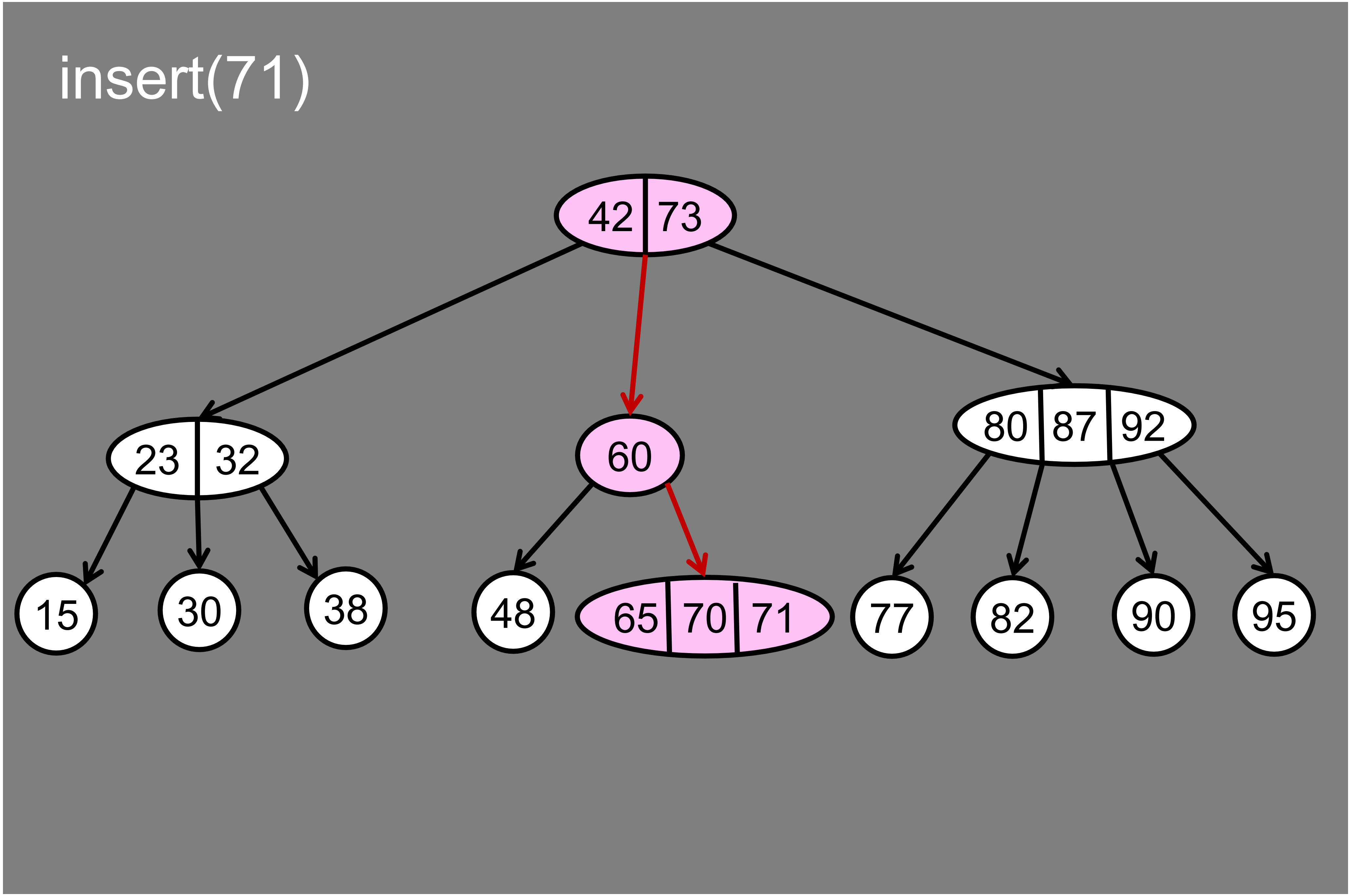
(2,4) Trees: Inserting



(2,4) Trees: Inserting

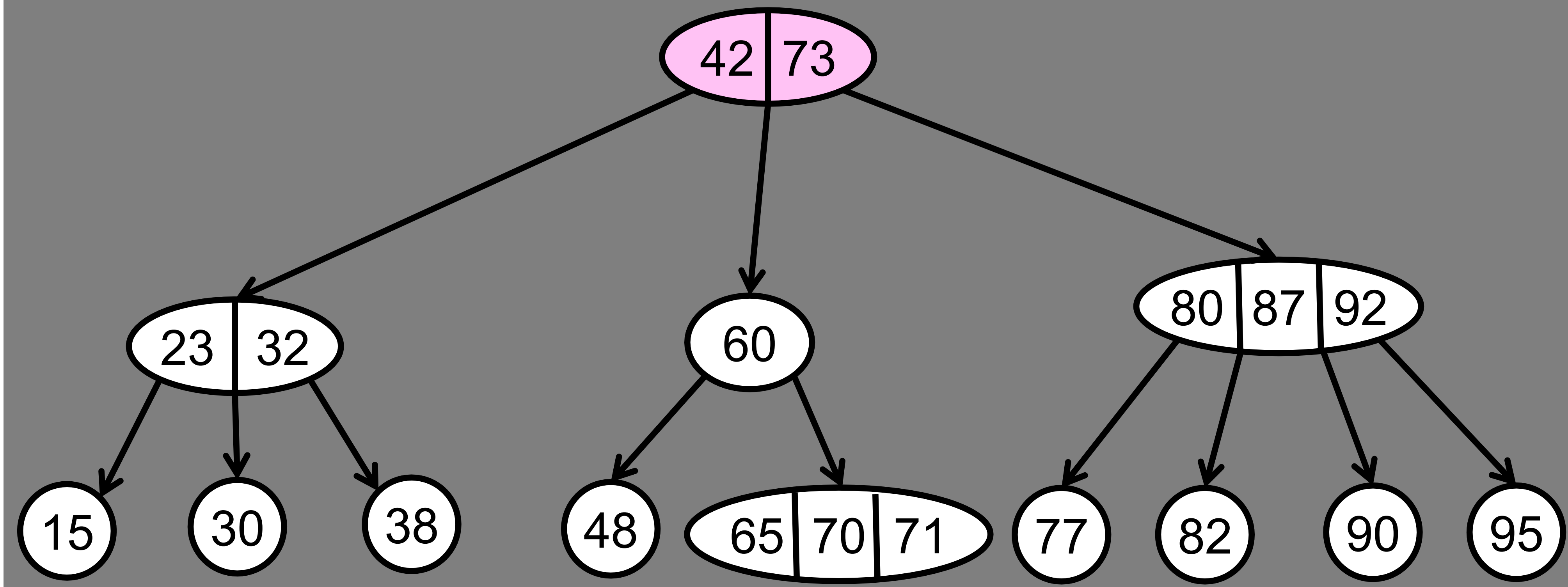


(2,4) Trees: Inserting

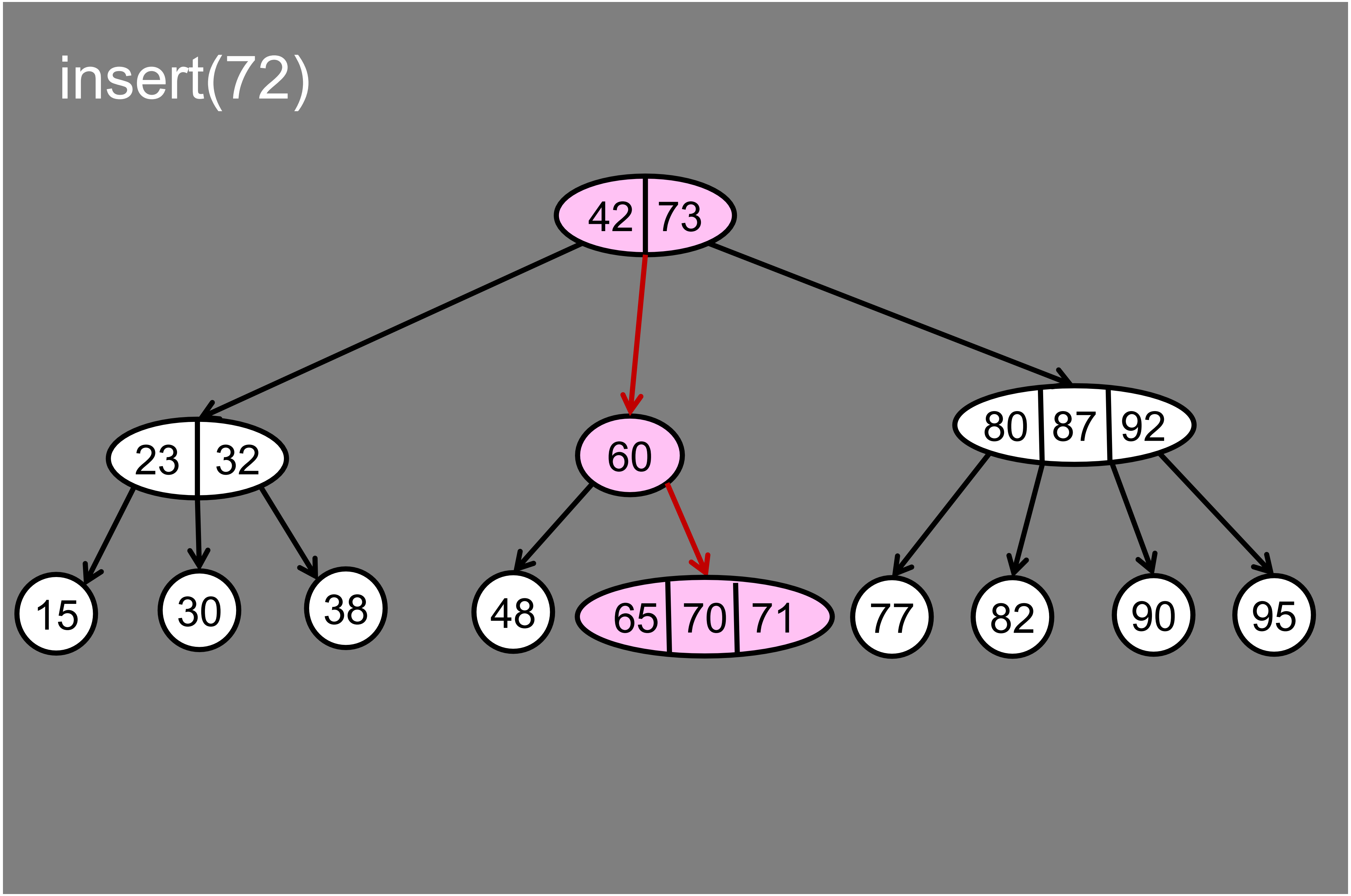


(2,4) Trees: Inserting

insert(72)

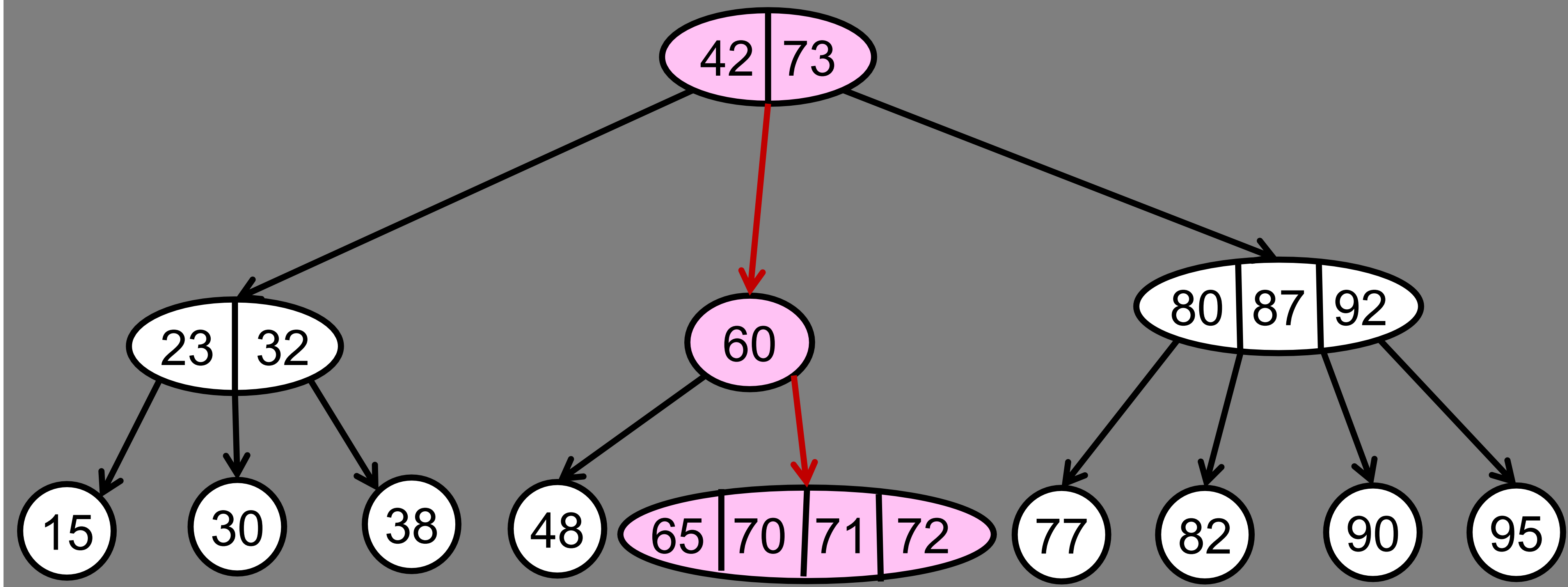


(2,4) Trees: Inserting



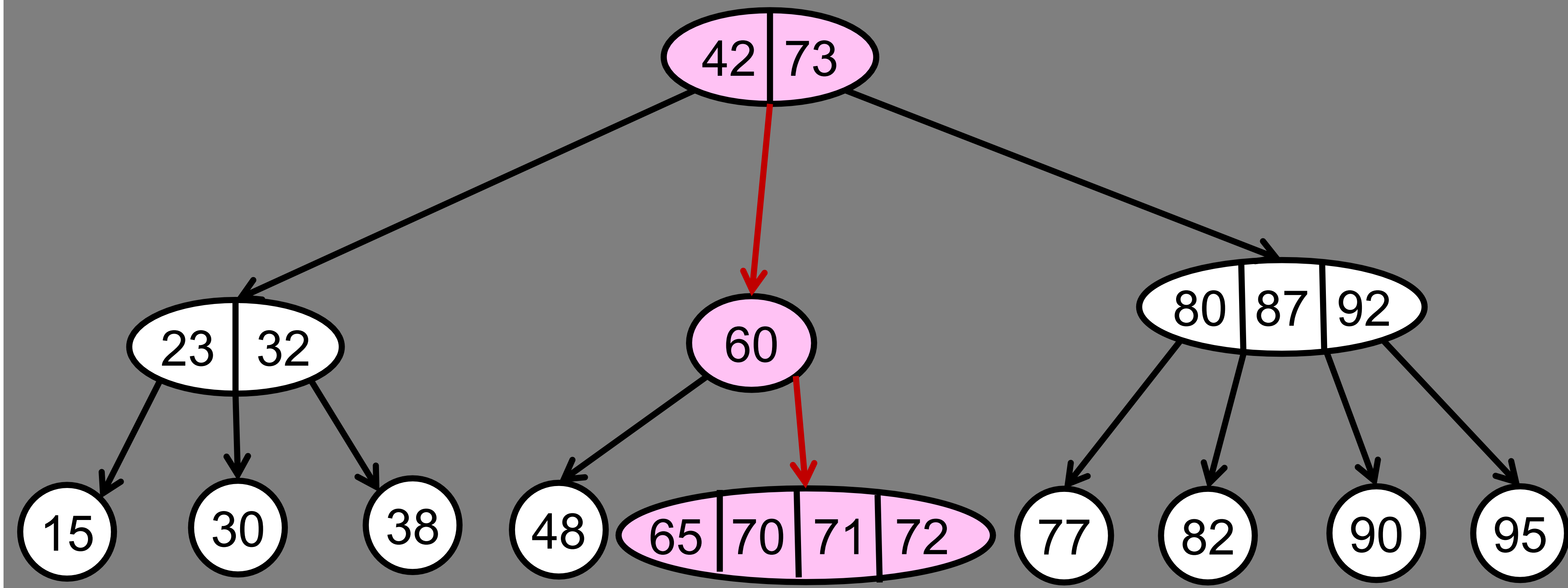
(2,4) Trees: Inserting

insert(72)



(2,4) Trees: Inserting

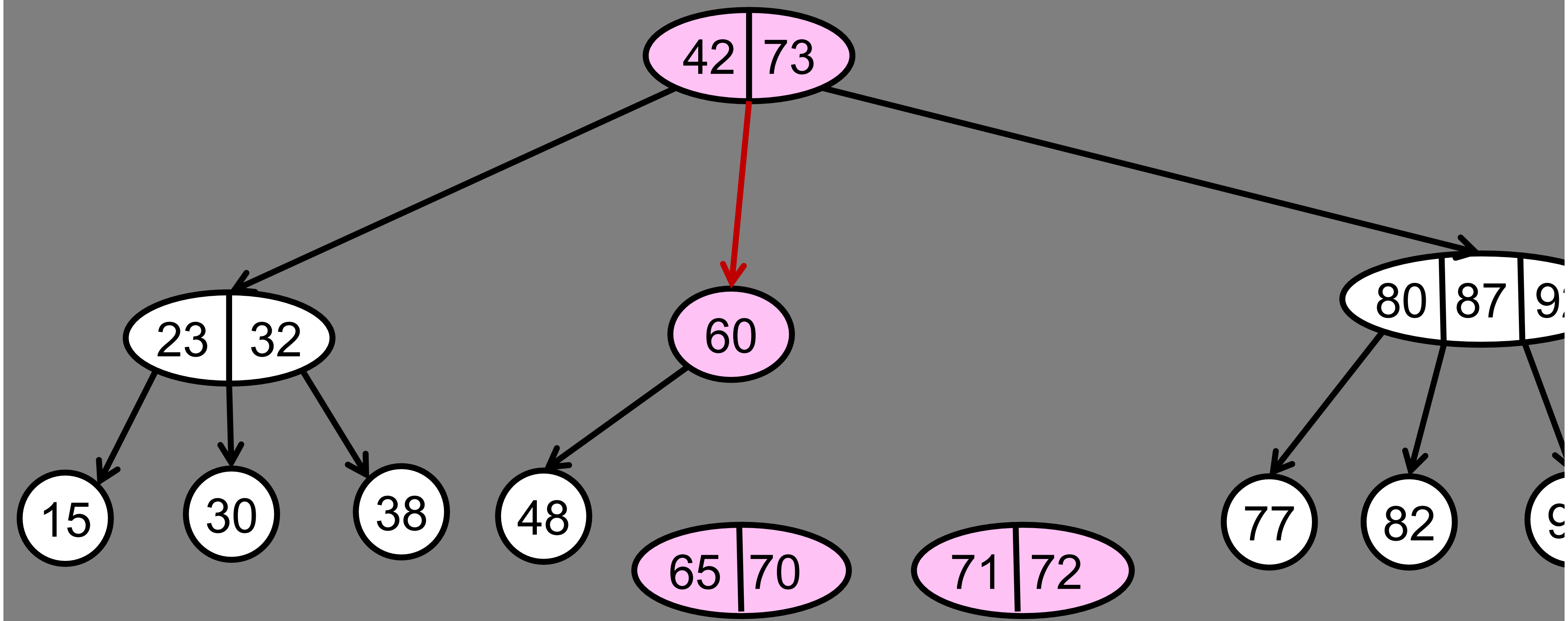
insert(72)



Too many (4) pivots…
Too many (5) children.

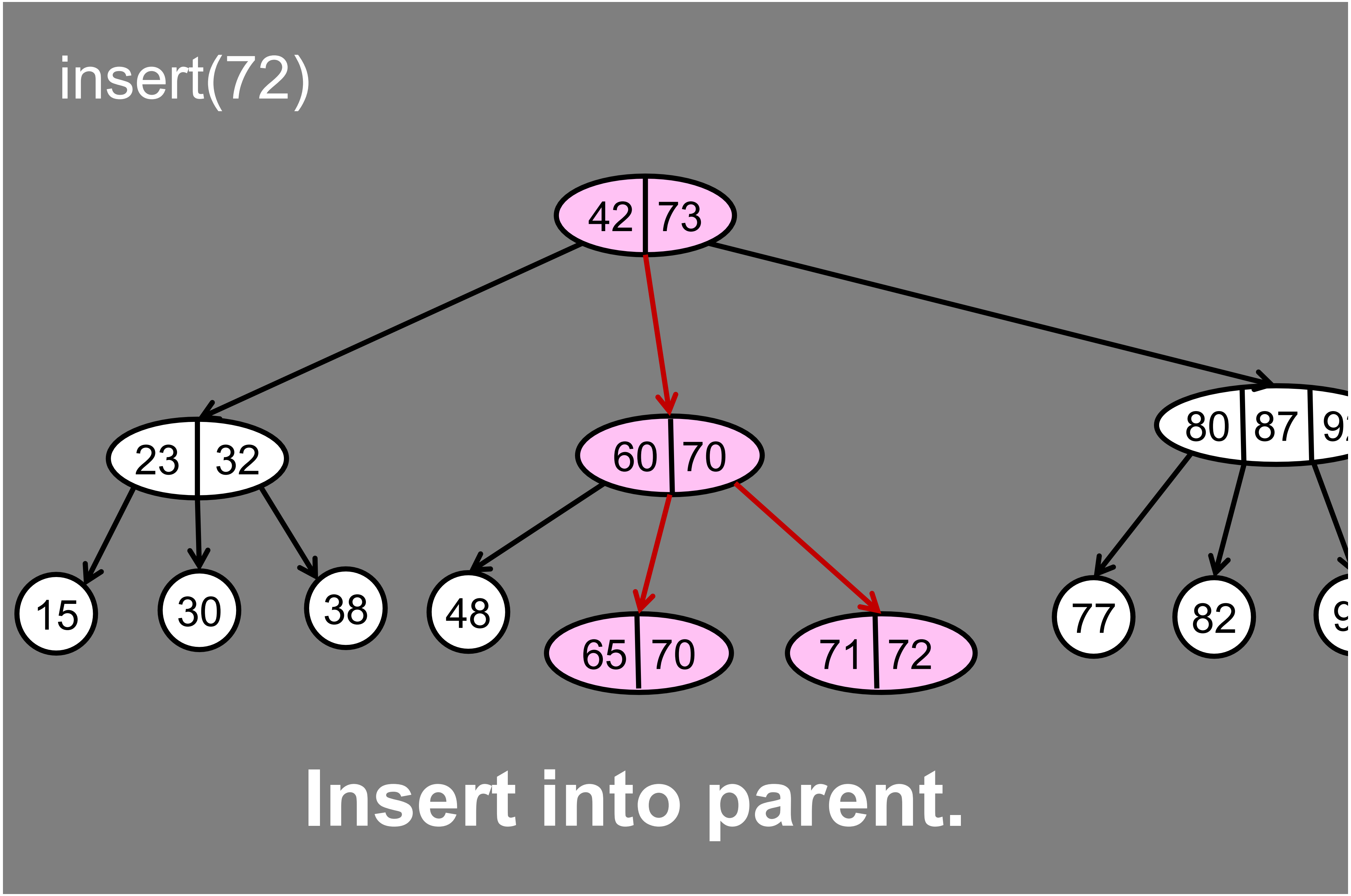
(2,4) Trees: Inserting

insert(72)



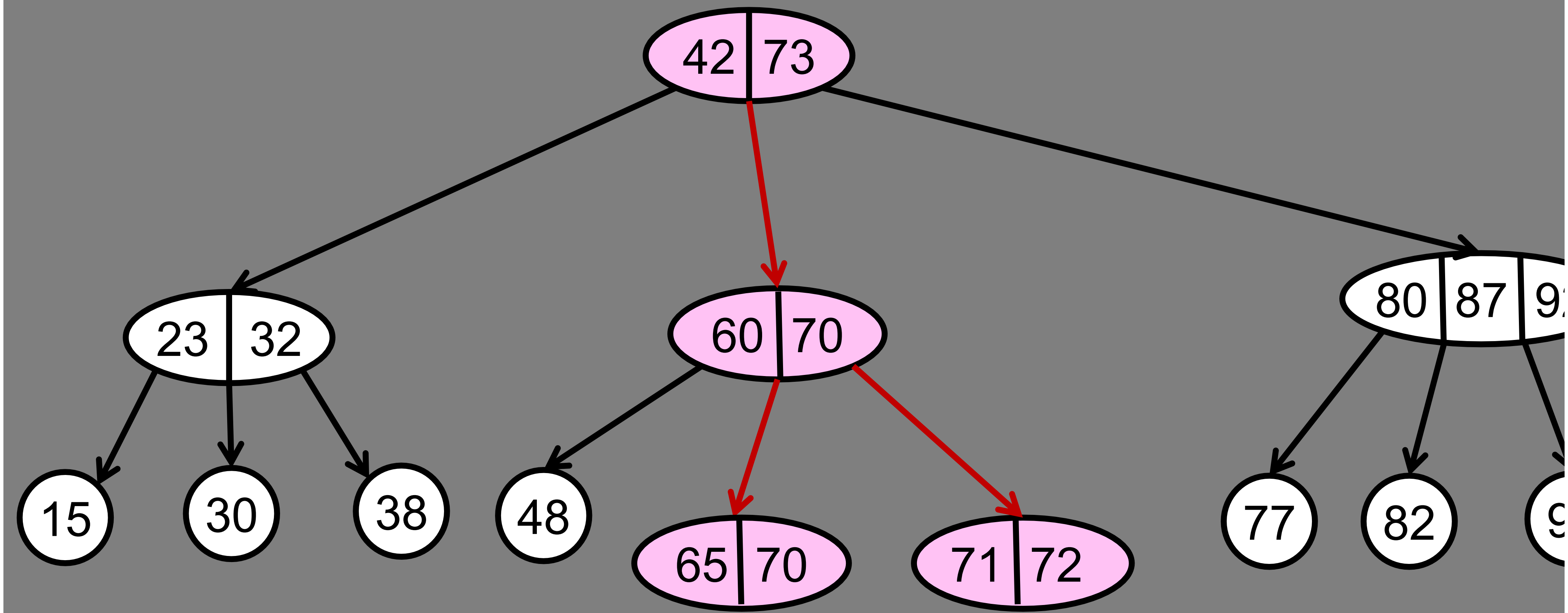
Split the node in two.

(2,4) Trees: Inserting



(2,4) Trees: Inserting

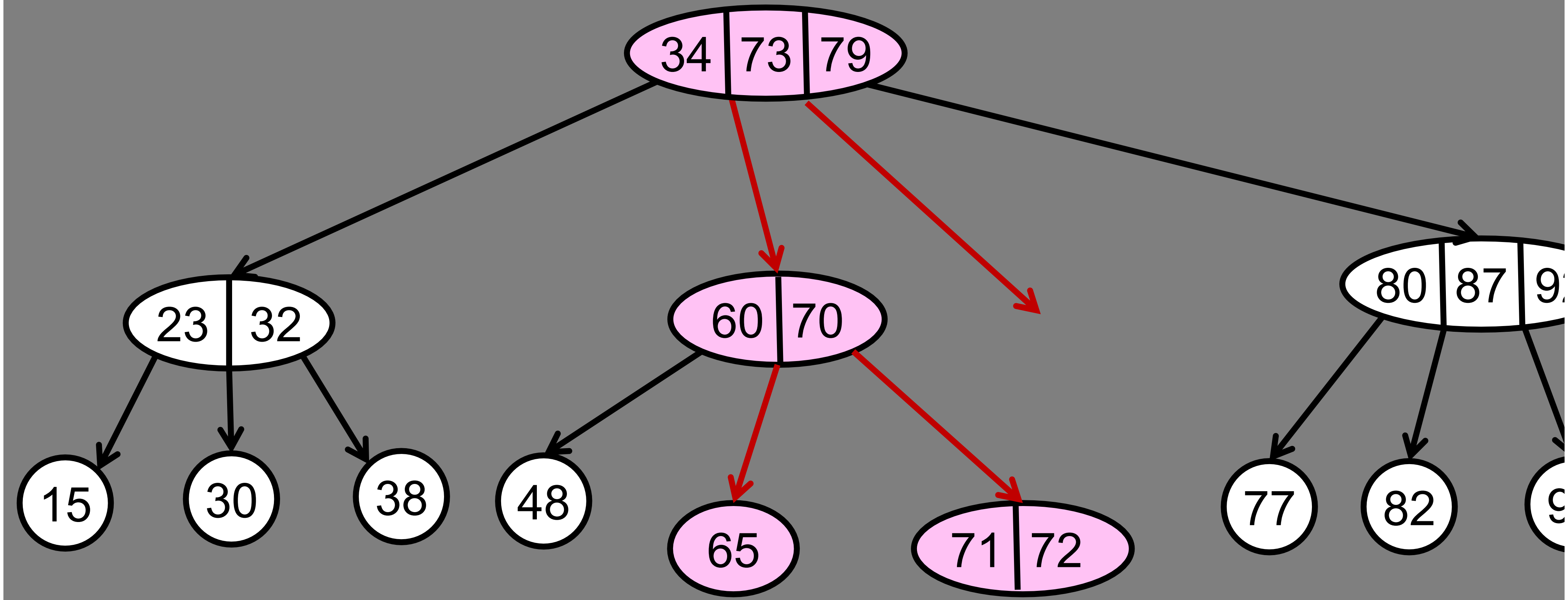
insert(72)



Recurse (if parent is full).

(2,4) Trees: Inserting

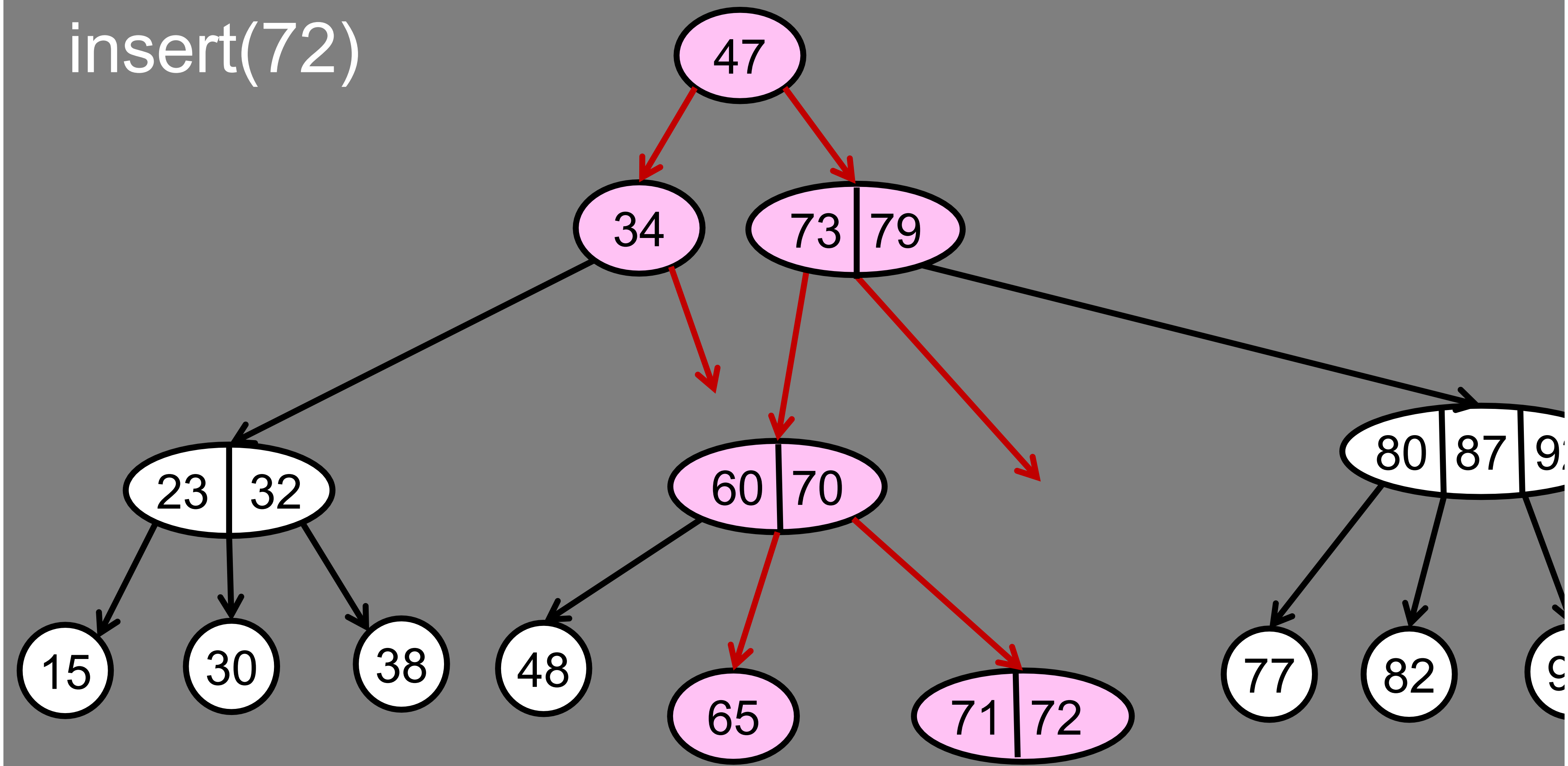
insert(72)



What if the root is full?

(2,4) Trees: Inserting

insert(72)



Split and create a new root.

(2,4) Trees

Key claim: preserves all properties:

1. Every node has [a,b] children.
2. Search tree property.
3. All leaves have the same depth.

(a,b)-Trees

Lazy option: do splitting when needed.

Proactive option: split in advance.

One pass insertion:

- If root contains b keys, split root and create new root.
- While searching for the leaf, split any node that is full (i.e., contains b keys).
- On arrival at leaf, there is enough space in the leaf to add the key!

(a, b)-trees

delete(k):

1. Search for leaf node v containing key k
2. Delete key k from leaf node v.
3. If v is root and has only one child, delete root.
4. If $|v| < a$:
   - Let u be a sibling of v.
   - Case 1: $|u| + |v| > b-1$

     Divide keys evenly between u and v.

     Each gets at least $b/2 \geq a$.
   - Case 2: $|u| + |v| \leq b-1$

     Merge u and v.

     Recursively delete pivot from parent.

     Update parent/child pointers.

B-tree

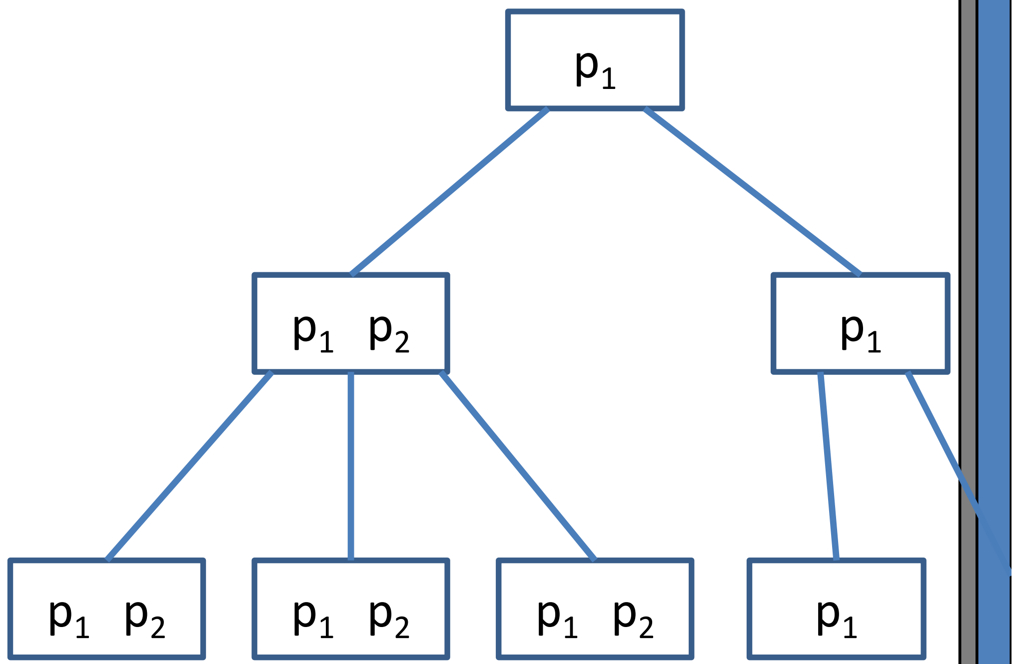
Fix a and b:

Set a = B, b = 2B.

Performance:

- Reading/writing each node of the tree takes O(1) block transfers.
- Insert/delete requires reading/writing O(1) nodes at each level of the tree.
- Thus the total cost of each read/write operation is:

$$O(\log_B n)$$



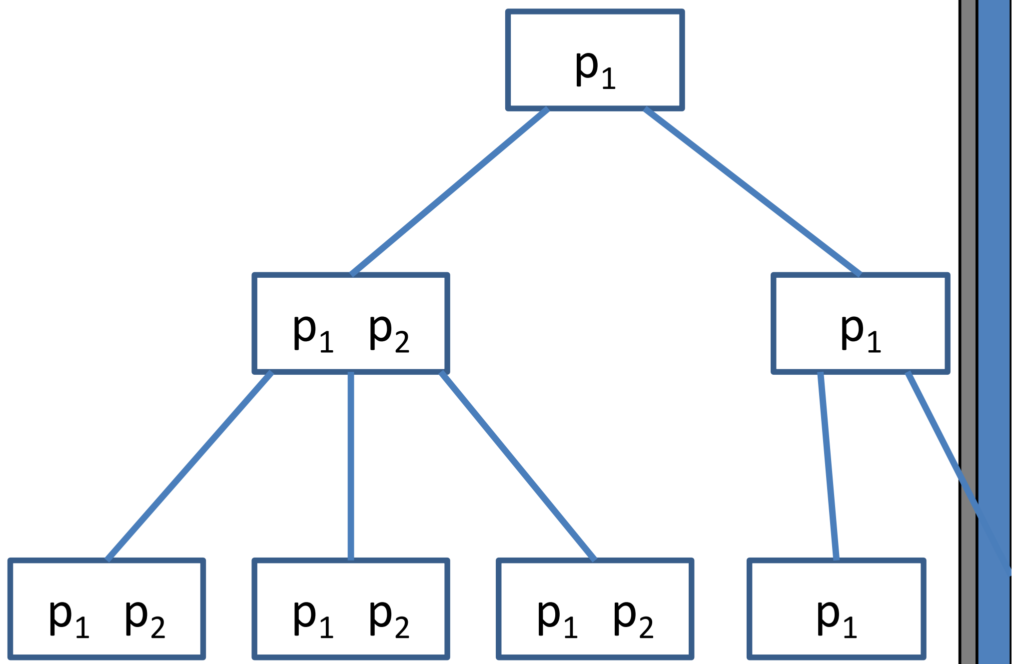
B-tree

Fix a and b:

Set a = B, b = 2B.

Some numbers:

- Assume your disk has 16 KB sized blocks.
- Assume you have 10 TB database.
- Then your B-tree has 3 levels.
- Since the root and first level are always in cache (e.g., 256 MB), each operation requires 1 cache miss.



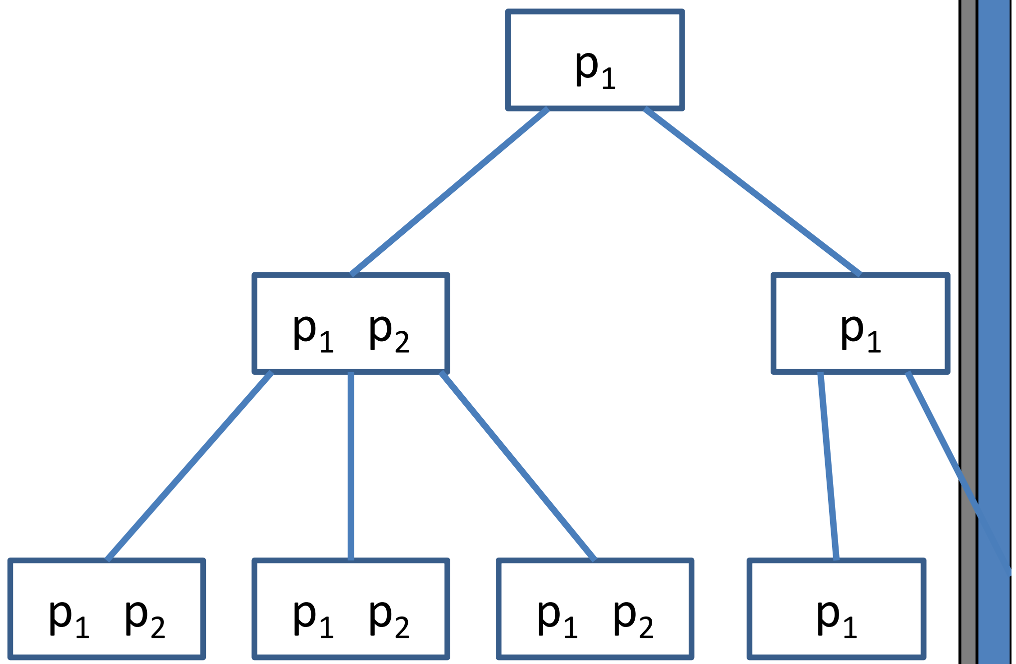
B-tree

Fix a and b:

Set $a = B$, $b = 2B$.

Some numbers:

- Assume your disk has 16 KB sized blocks.
- Assume you have 1000 TB database.
- Then your B-tree has 4 levels.
- Since the root and first level are always in cache (e.g., 256 MB), each operation requires 2 cache miss.



Amortized Analysis

Fix a and b:

Set $a = B$, $b = 5B$.

How often does a node split or merge?

About to split:

$p_1 \quad p_2 \quad p_3 \quad \ldots \quad p_{b-1} \quad p_b$

About to merge:

$p_1 \quad p_2 \quad p_3 \quad \ldots \quad p_{a-1} \quad p_a$

Amortized Analysis

Fix a and b:

Set a = B, b = 5B.

Claim:

After each split, a node has:

- ≥ 2B-1 keys
- ≤ 4B keys

Because:

- b/2 = (5/2)B > 2B
- b/2 = (5/2)B < 5B

About to split:

p_1 p_2 p_3 … p_{b-1} p_b

About to merge:

p_1 p_2 p_3 … p_{a-1} p_a

Amortized Analysis

Fix a and b:

Set $a = B$, $b = 5B$.

Claim:

After each share/merge, a node has:

$\geq 2B-1$ keys

$\leq 4B$ keys

Modify share/merge rule:

- If $|u| + |v| > 4B$, share.
- If $|u| + |v| \leq 4B$, merge.

About to split:

$\boxed{p_1 \; p_2 \; p_3 \; \ldots \; p_{b-1} \; p_b}$

About to merge:

$\boxed{p_1 \; p_2 \; p_3 \; \ldots \; p_{a-1} \; p_a}$

$\boxed{(B + 5B)/2 < 4B}$

$\boxed{(B-1 + B) \geq 2B-1}$

Amortized Analysis

Fix a and b:

Set $a = B$, $b = 5B$.

Claim:

After each split/share/merge, a node has:

$\geq 2B-1$ keys

$\leq 4B$ keys

About to split:

p_1 p_2 p_3 … p_{b-1} p_b

About to merge:

p_1 p_2 p_3 … p_{a-1} p_a

Amortized Analysis

Fix a and b:

Set $a = B$, $b = 5B$.

Claim:

After each split/share/merge, a node has:

$\geq 2B-1$ keys

$\leq 4B$ keys

How long until next split/share/merge?

About to split:

p_1 p_2 p_3 … p_{b-1} p_b

About to merge:

p_1 p_2 p_3 … p_{a-1} p_a

Amortized Analysis

Fix a and b:
Set a = B, b = 5B.

Claim:
After each split/share/merge, a node has:

≥ 2B-1 keys

≤ 4B keys

How long until next split/share/merge?
At least B-1 more operations.

About to split:

p_1 p_2 p_3 … p_{b-1} p_b

About to merge:

p_1 p_2 p_3 … p_{a-1} p_a

Amortized Analysis

Fix a and b:
Set a = B, b = 5B.

Claim:
After each split/share/merge, at least B-1 more operations before the next split/share merge.

Claim:
The amortized cost of split/share/merge is $O(1/B)$ per node, and $O((1/B)\log_B(B))$ per operation.

About to split:

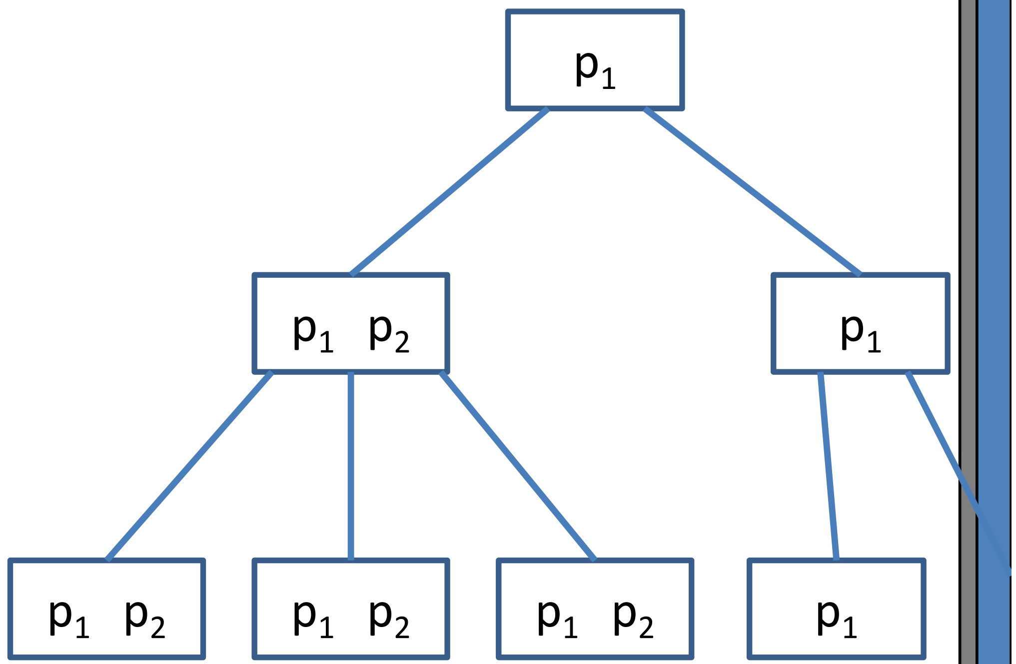
p_1 p_2 p_3 … p_{b-1} p_b

About to merge:

p_1 p_2 p_3 … p_{a-1} p_a

B-tree

What changes if each node stores a parent pointer?



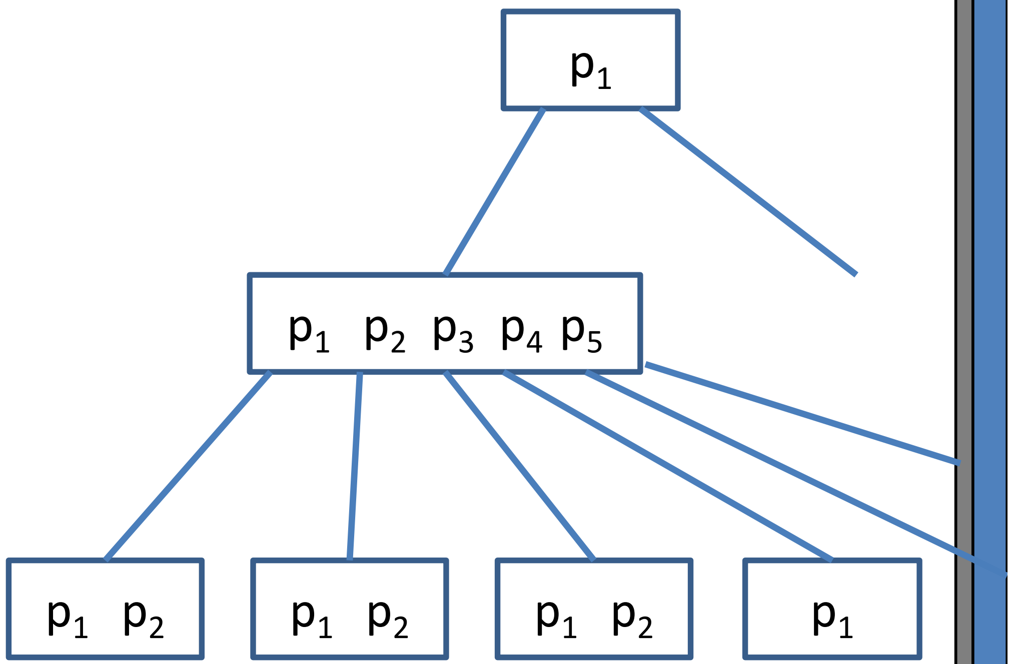
B-tree

What changes if each node stores a parent pointer?

On every split, need to update the parent pointer for $\Theta(B)$ children!

Very expensive!

Insert may cost $\Theta(B \log_B n)$ if every level needs to be split!



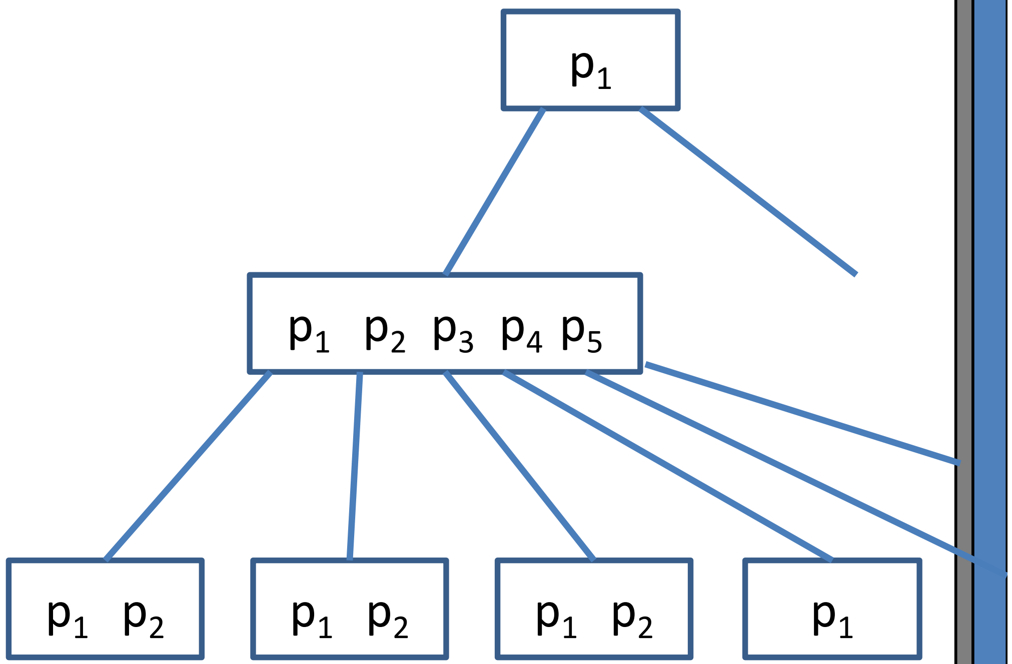
B-tree

What changes if each node stores a parent pointer?

On every split, need to update the parent pointer for $\theta(B)$ children!

NOT very expensive (amortized)!

Splitting/merging may cost $\theta((1/B)B \log_B n)$ amortized, if every level needs to be split!



Same for merging… Also helps with concurrency and locking…

Today's Plan

Searching and Sorting

1. B-trees
   - ⇒ Algorithm
   - ⇒ Amortized analysis

2. Buffer trees
   - ⇒ Write-optimized data structures
   - ⇒ Buffered data structures
   - ⇒ Amortized analysis

3. van Emde Boas Search Tree
   - ⇒ Cache-oblivious algorithms
   - ⇒ van Emde Boas memory layout

Cost Trade-Offs

Can you do better than a B-tree?

$\Rightarrow$ Is $O(\log_B n)$ optimal or can you do better?

Cost Trade-Offs

Can you do better than a B-tree?

$\Rightarrow$ Is $O(\log_B n)$ optimal or can you do better?

For searching, $O(\log_B n)$ is optimal.

(in the comparison-based model)

Exercise: prove it.

Cost Trade-Offs

Can you do better than a B-tree?

$\Rightarrow$ Is $O(\log_B n)$ optimal or can you do better?

For searching, $O(\log_B n)$ is optimal.

(in the comparison-based model)

For inserting/deleting, it is <u>NOT</u> optimal.

(Example: linked list has O(1) inserts.)

Cost Trade-Offs

Goal:

A external memory data structure with fast searches, and super-fast insertions/deletions.

"Write-optimized data structure."

Cost Trade-Offs

Why?

1) Some applications have more update operations than query operations (e.g., logs).

2) Some applications have *a lot* of update operations, so it pays to make them faster.

3) Some applications have expensive updates, e.g., a multi-index database.

Cost Trade-Offs

Multi-index database:

Database may have more than one index, e.g.:

- Employee database: name, age, salary, position

Advantage: search by any index in $O(\log_B n)$ time.

Disadvantage: cost of an update?

Cost Trade-Offs

Multi-index database:

Database may have more than one index, e.g.:

- Employee database: name, age, salary, position

Advantage: search by any index in $O(\log_B n)$ time.

Disadvantage: cost of an update: $O(k\log_B n)$ time for k indices.

Ideal trade-off: update should be k-times faster than searches!

Streaming a Graph

Data arrives in a stream: $S = s_1, s_2, \ldots, s_T$

Each s_j is an edge in the graph.

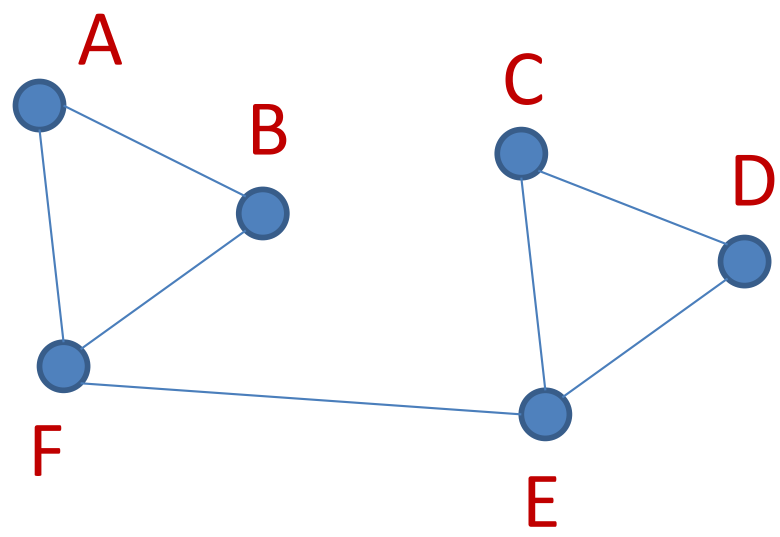
- ⇒ Each edge shows up exactly once.
- ⇒ Edges show up in an arbitrary (worst-case) order.

Example:

S = (A,B), (C,D), (F,E), (C,E), (E,D), (A,F), (B,F)

Goal: minimize space

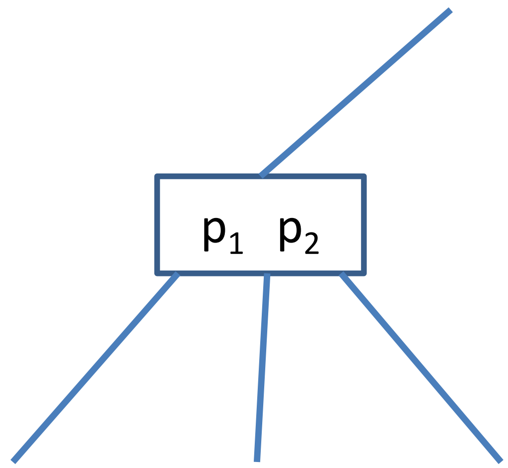
- ⇒ Sublinear space is often impossible.
- ⇒ Best possible: O(n log n) space.
- ⇒ Focus on dense graphs.



Buffer Tree

Recipe:

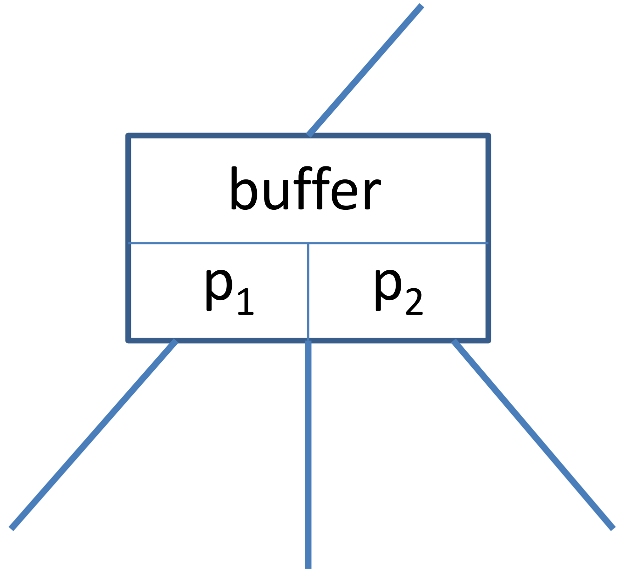
1) Build a (2,4)-tree.



Buffer Tree

Recipe:

1) Build a (2,4)-tree.
2) Add a buffer of size 2B to every node in the tree.



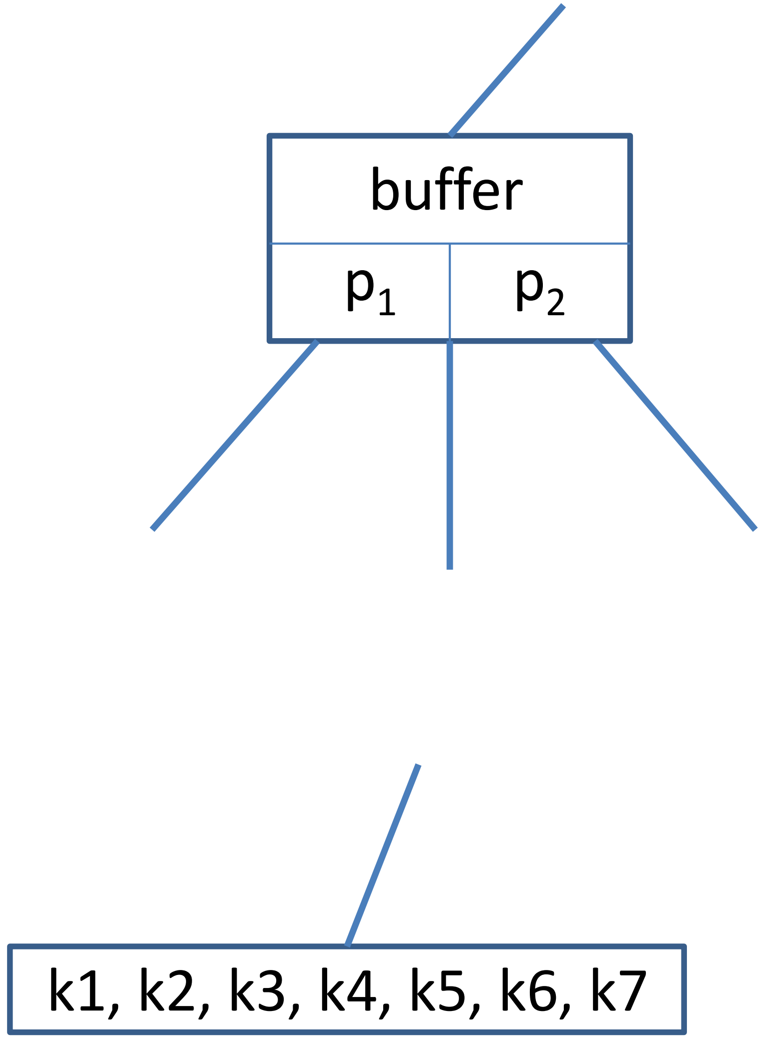
Buffer Tree

Recipe:

1) Build a (2,4)-tree.

2) Add a buffer of size 2B to every node in the tree.

3) For each leaf, ensure it has ≥ B keys and ≤ 5B keys



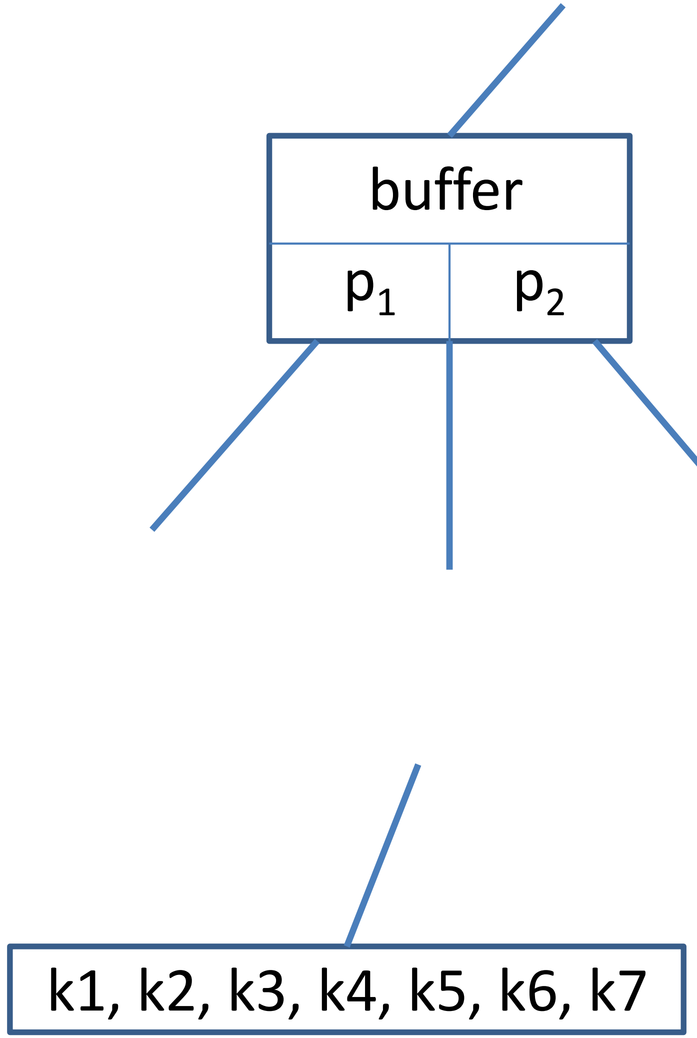
Buffer Tree

insert(key):

1) Add ins[key] to root buffer.

2) Stop.

Cost: O(1)



Buffer Tree

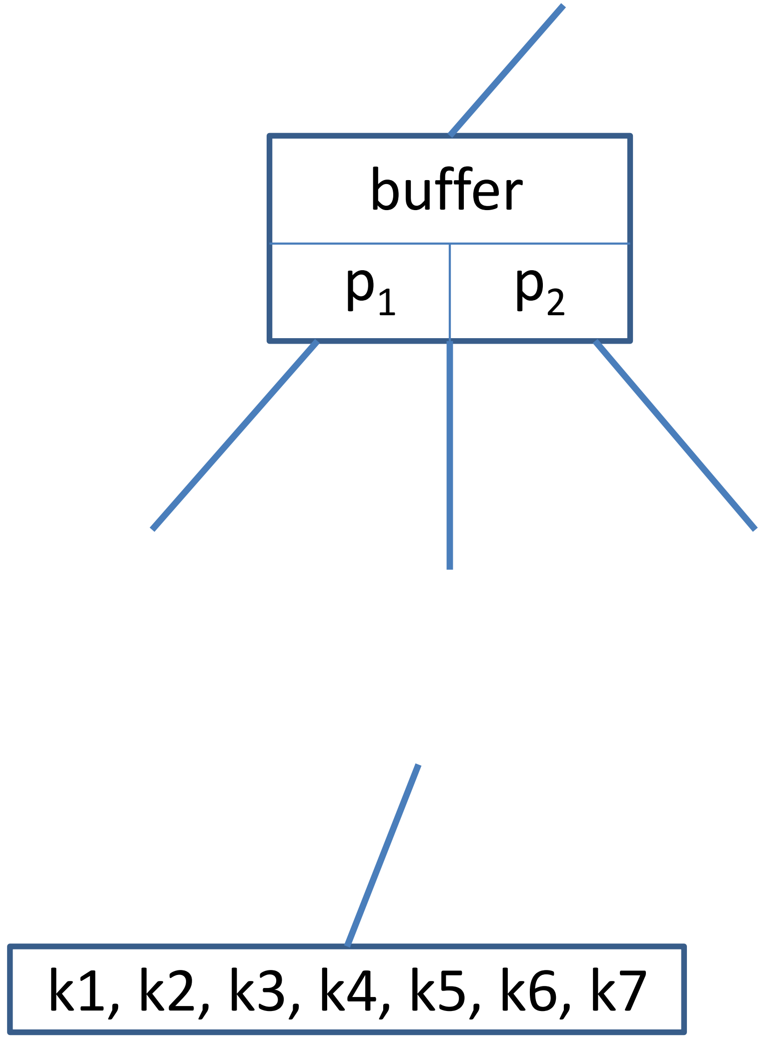
insert(key):

1) Add ins[key] to root buffer.

2) Clean buffer:
   - If del[key] is in buffer, remove it.
   - Remove duplicate ins[key] operations.

3) Stop.

Cost: O(1)



Buffer Tree

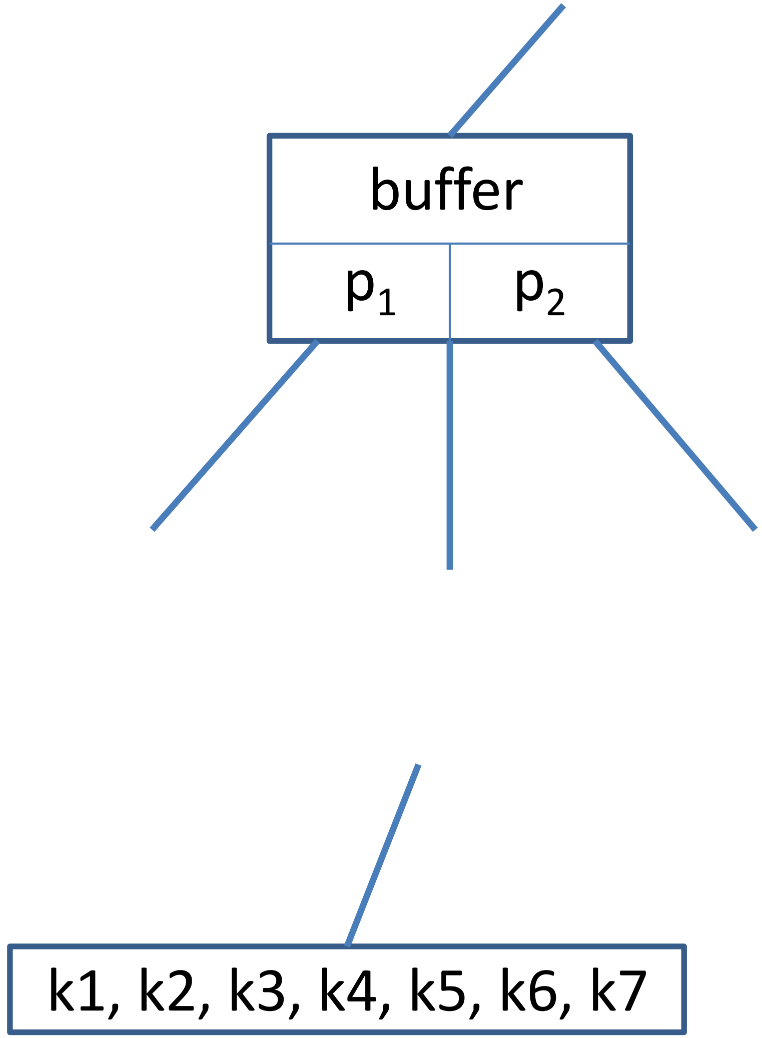
insert(key):

1) Add ins[key] to root buffer.

2) Clean buffer:
   - If del[key] is in buffer, remove it.
   - Remove duplicate ins[key] operations.

3) If |buffer| > B, then flush the buffer.

Cost: O(1) + buffer flush



Buffer Tree

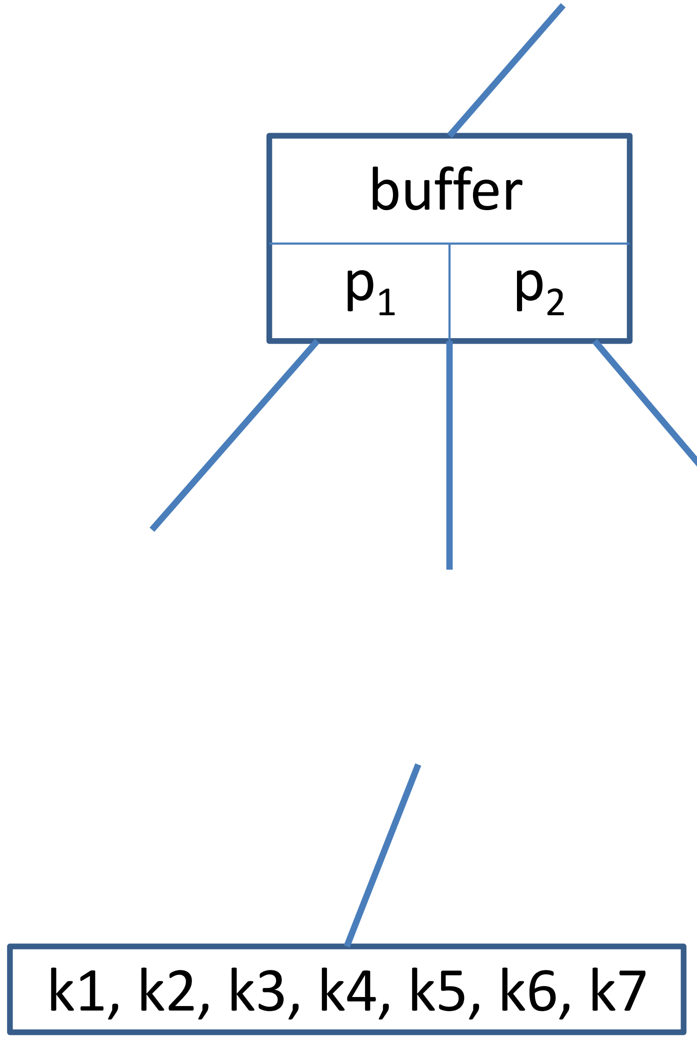
delete(key):

1) Add del[key] to root buffer.

2) Clean buffer:
   - If ins[key] is in buffer, remove it and del[key].
   - Remove duplicate del[key] operations.

3) If |buffer| > B, then flush the buffer.

Cost: O(1) + buffer flush



Buffer Tree

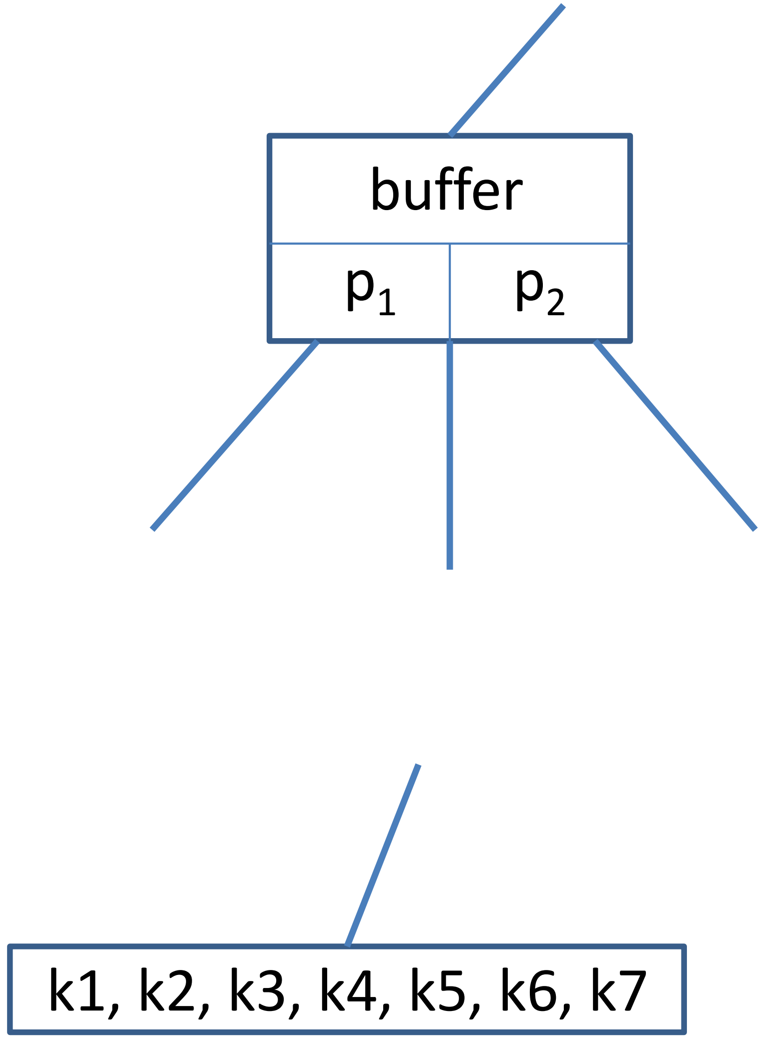
search(key):

1) Perform a tree walk from root to leaf.

2) At every node on the walk, search the buffer for the key.

3) When you get to a leaf, search the leaf for the key.

Cost?



Buffer Tree

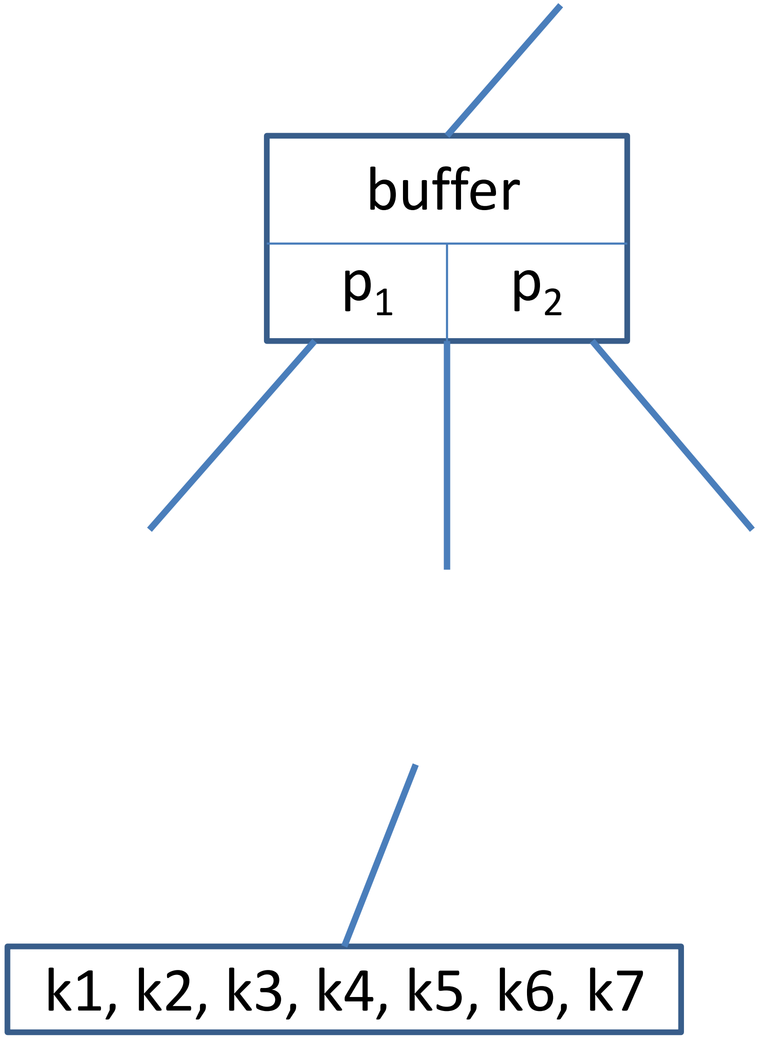
search(key):

1) Perform a tree walk from root to leaf.

2) At every node on the walk, search the buffer for the key.

3) When you get to a leaf, search the leaf for the key.

Cost: $O(\log n)$



Notice branching factor: 2, not B.

Buffer Tree

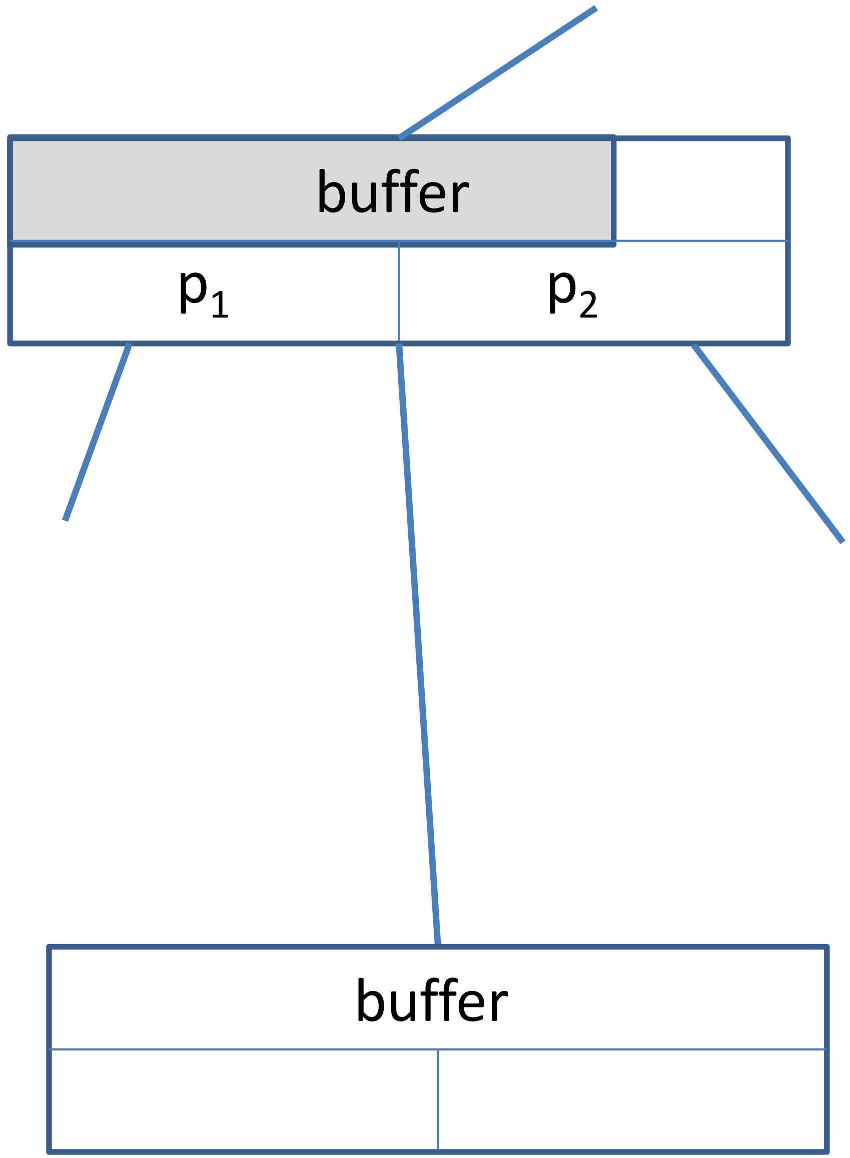
flush(node v):

1) Sort the buffer.

2) Move every operation to its proper child.

3) Clean child buffer (e.g., remove duplicates).

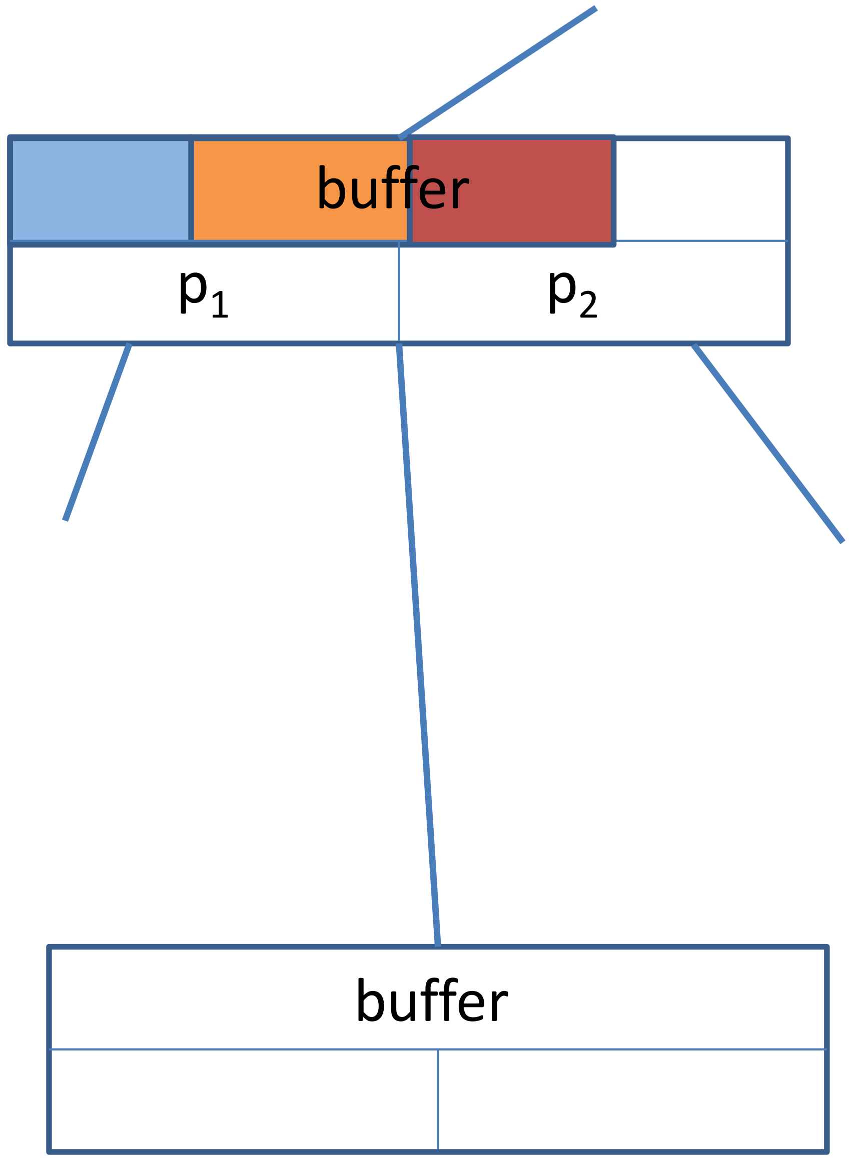
4) Recursively flush child buffer, if necessary.



Buffer Tree

flush(node v):

1) Sort the buffer.
2) Move every operation to its proper child.
3) Clean child buffer (e.g., remove duplicates).
4) Recursively flush child buffer, if necessary.

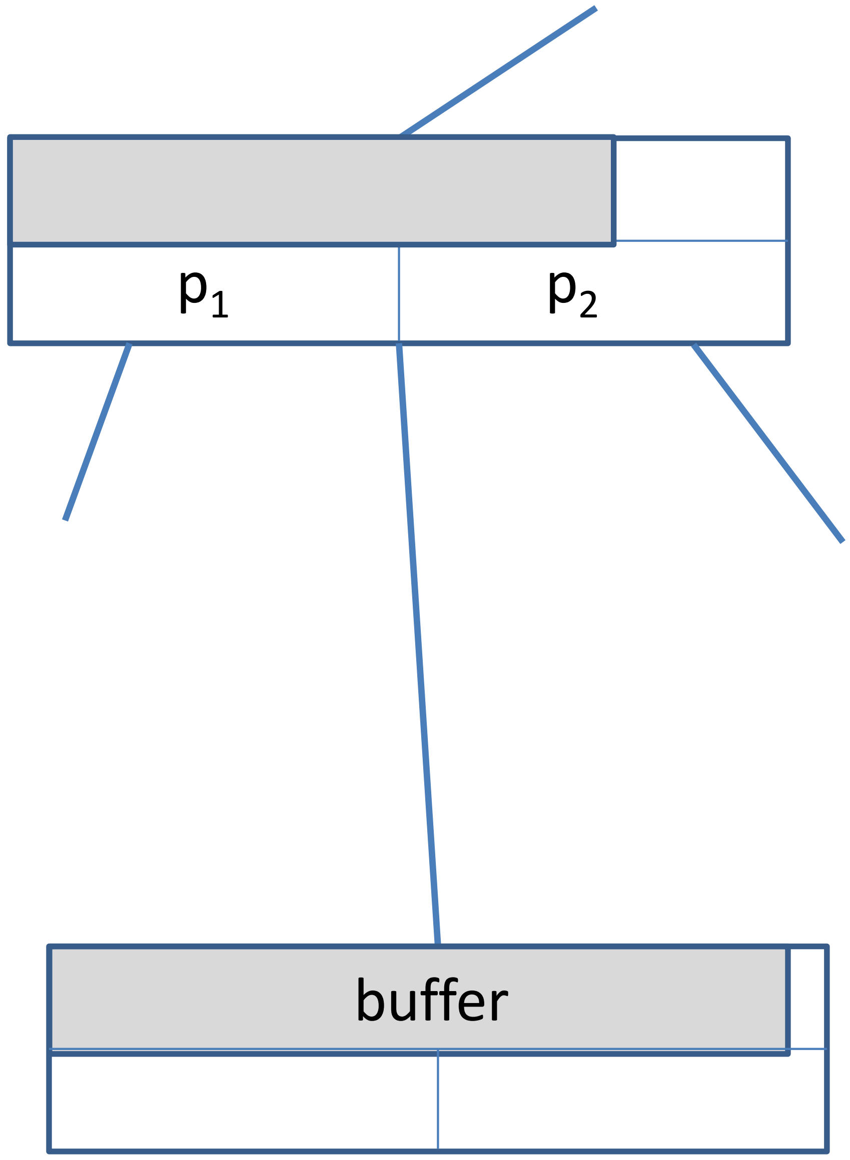


Buffer Tree

One special case:

If child buffer gets full, then pause, flush it, and then continue flushing the parent.

- Each flush is size at most 2B.
- So only need to pause for flushing each child once.

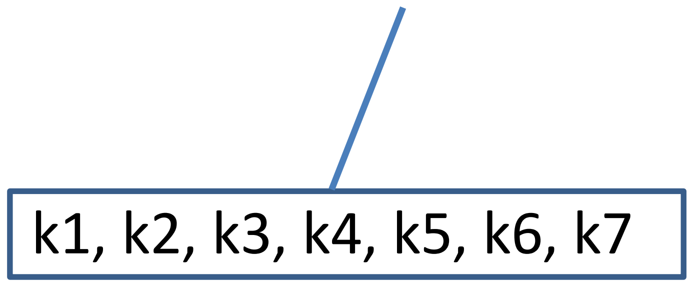


Buffer Tree

At a leaf

When flushing to a leaf:

- A leaf has no buffer.
- All keys stored at leaves.

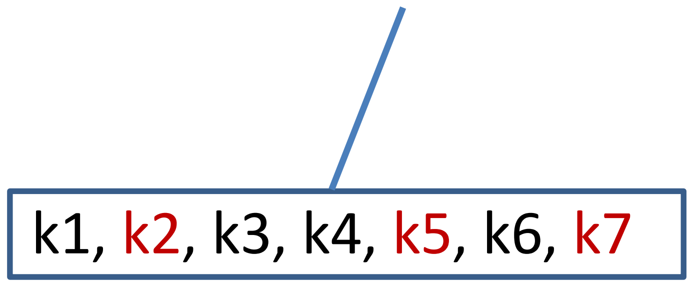


Buffer Tree

At a leaf

When flushing to a leaf:

- A leaf has no buffer.
- All keys stored at leaves.
- First perform all delete operations at leaf.

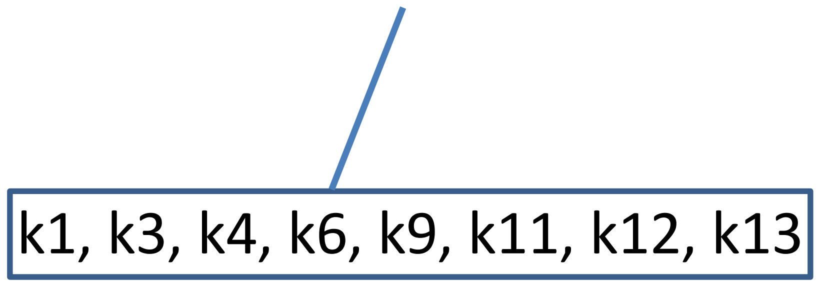


Buffer Tree

At a leaf

When flushing to a leaf:

- A leaf has no buffer.
- All keys stored at leaves.
- First perform all delete operations at leaf.
- Then perform inserts, and do splits as needed.

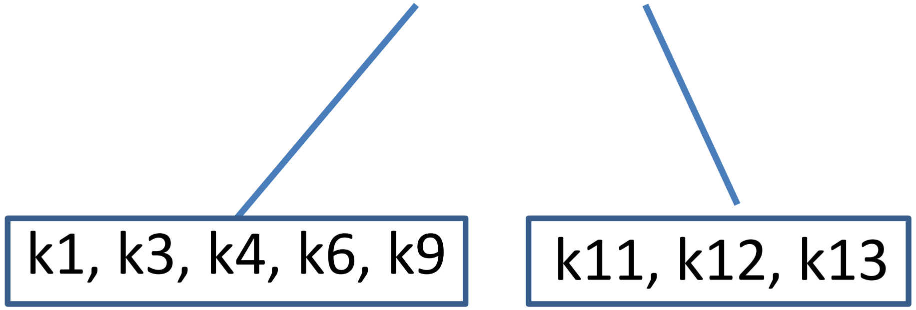


Buffer Tree

At a leaf

When flushing to a leaf:

- A leaf has no buffer.
- All keys stored at leaves.
- First perform all delete operations at leaf.
- Then perform inserts, and do splits as needed.



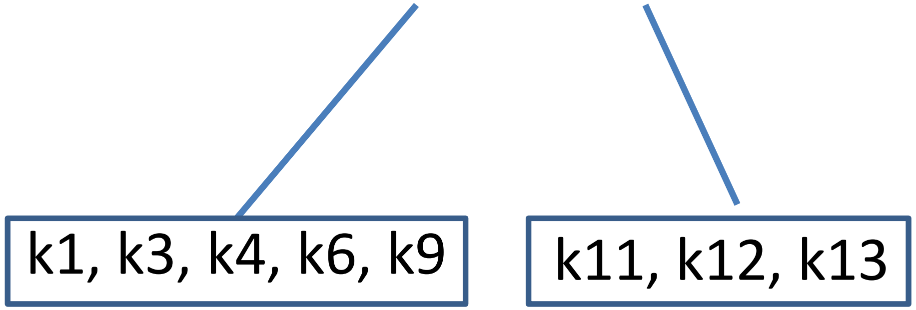
splitting buffers is easy…

Buffer Tree

At a leaf

When flushing to a leaf:

- A leaf has no buffer.
- All keys stored at leaves.
- First perform all delete operations at leaf.
- Then perform inserts, and do splits as needed.
- At end, do merges.



merging buffers is easy… but can result in flush operations

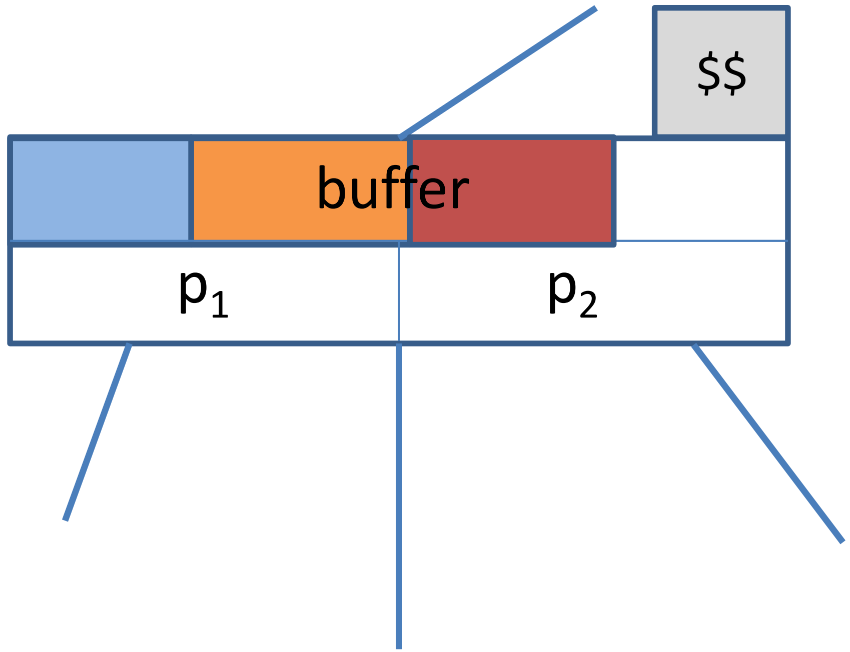
Buffer Tree

Amortized Analysis

Each node has a bank account.

Every operation:

- If root-to-leaf path for a key touches a node, it adds θ(1/B) dollars to the bank account for that node.



Buffer Tree

Amortized Analysis

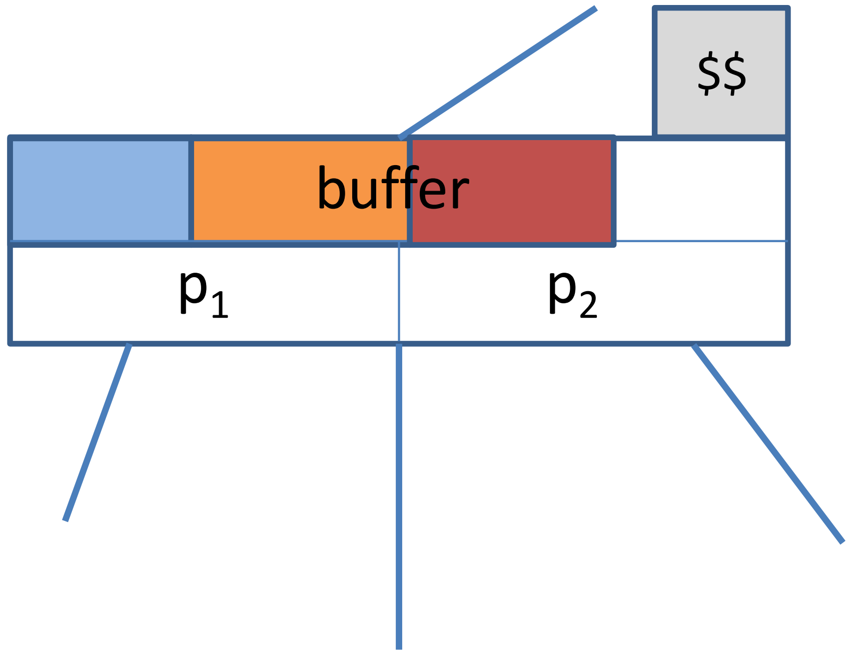
Each node has a bank account.

Every operation:

- If root-to-leaf path for a key touches a node, it adds $\theta(1/B)$ dollars to the bank account for that node.

Cost: $O(1)$ + buffer flush costs

➔ Pay for buffer flush from bank account.

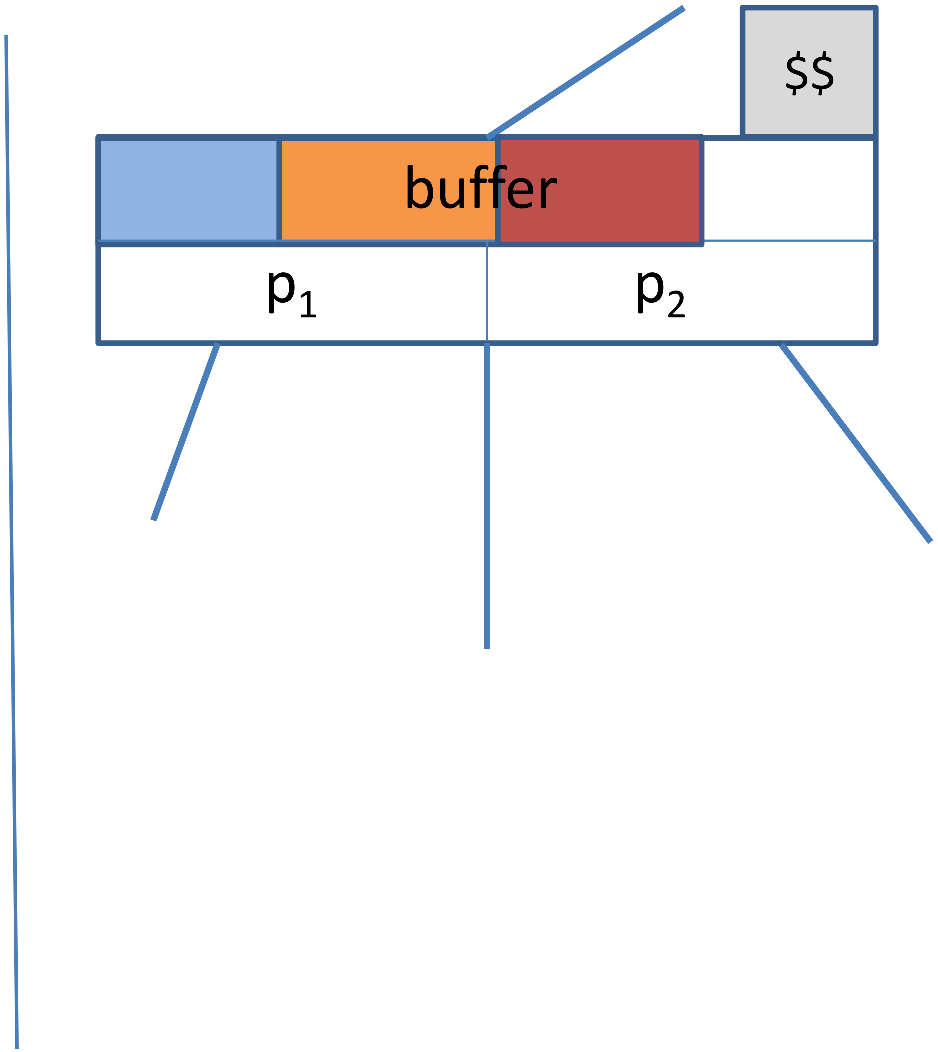


Buffer Tree

Amortized Analysis

Cost of flush at node v:

1. Load the buffer and pointers: O(1)

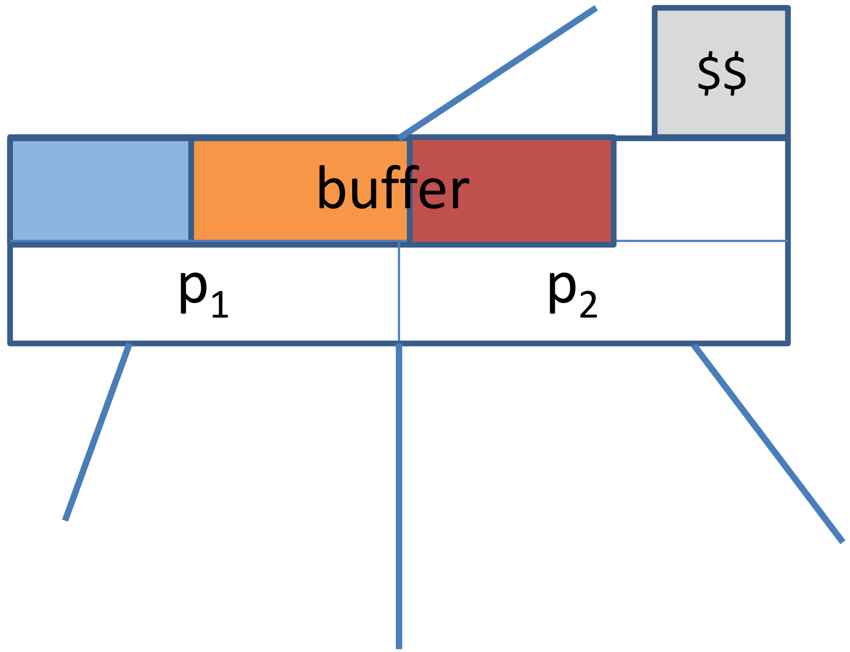


Buffer Tree

Amortized Analysis

Cost of flush at node v:

1. Load the buffer and pointers: O(1)
2. Sort the buffer: free.

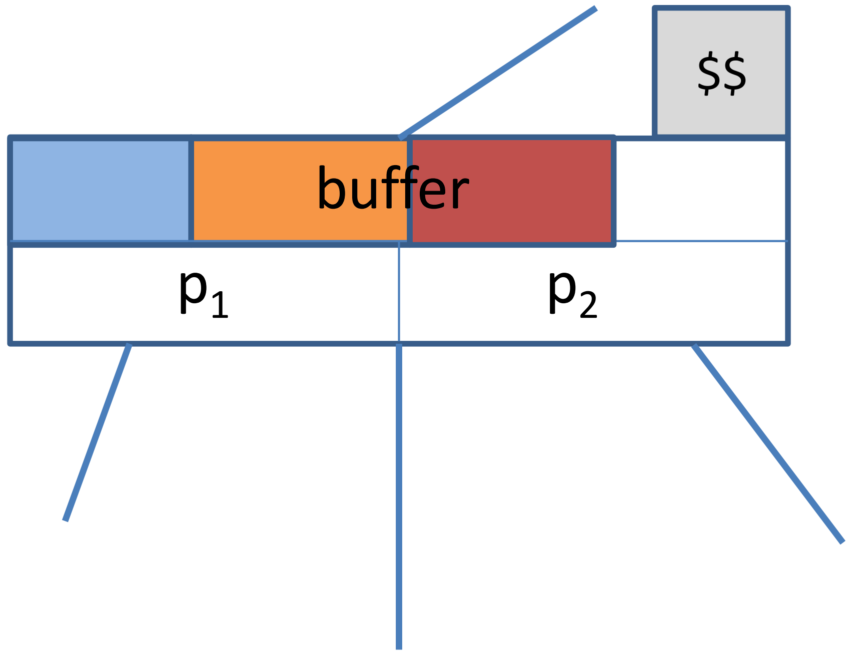


Buffer Tree

Amortized Analysis

Cost of flush at node v:

1. Load the buffer and pointers: O(1)
2. Sort the buffer: free.
3. Partition the keys among the children: free.

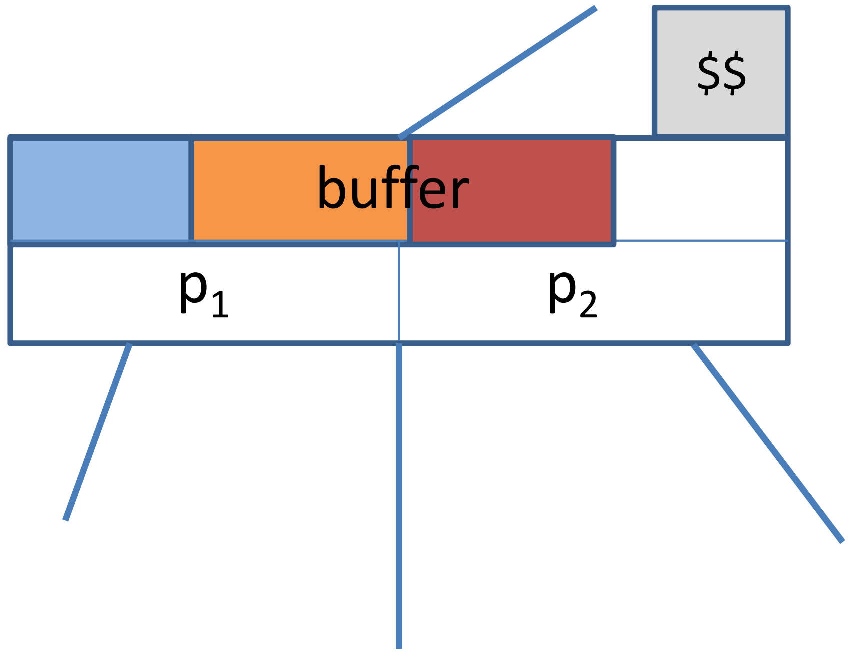


Buffer Tree

Amortized Analysis

Cost of flush at node v:

1. Load the buffer and pointers: O(1)
2. Sort the buffer: free.
3. Partition the keys among the children: free.
4. Load the buffers of the child nodes: O(1)
5. Move keys to child buffers: free.

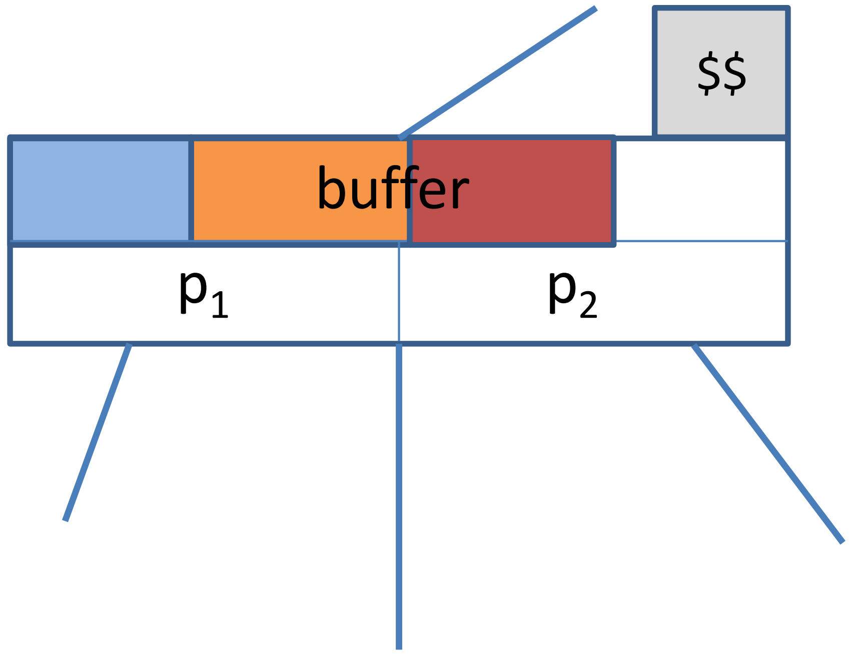


Buffer Tree

Amortized Analysis

Cost of flush at node v:

1. Load the buffer and pointers: O(1)
2. Sort the buffer: free.
3. Partition the keys among the children: free.
4. Load the buffers of the child nodes: O(1)
5. Move keys to child buffers: free.
6. Recursive flushing charged to child nodes.

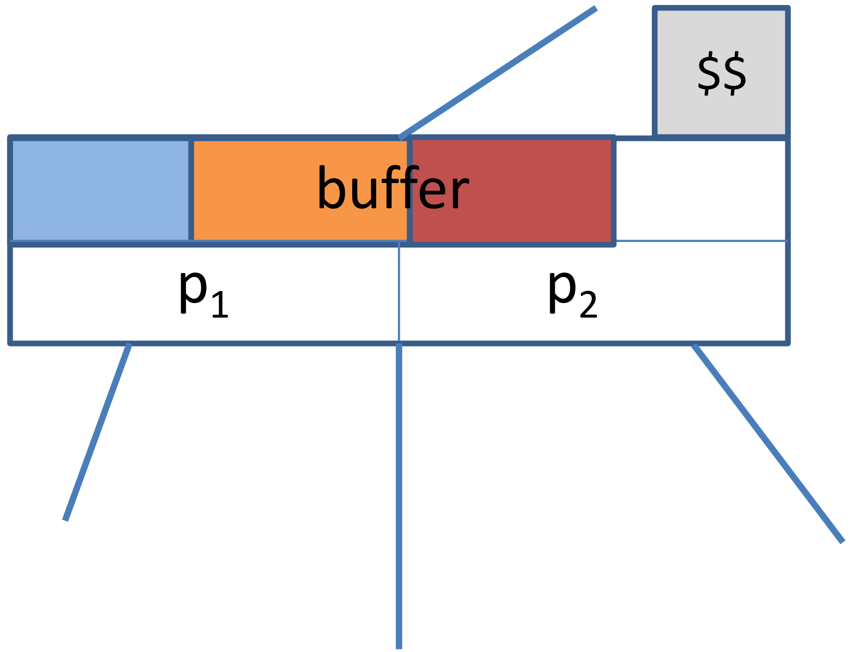


Buffer Tree

Amortized Analysis

Cost of flush at node v:

1. Load the buffer and pointers: O(1)
2. Sort the buffer: free.
3. Partition the keys among the children: free.
4. Load the buffers of the child nodes: O(1)
5. Move keys to child buffers: free.
6. Recursive flushing charged to child nodes.



Each flush costs O(1).

A flush only occurs when buffer contains at least B items.

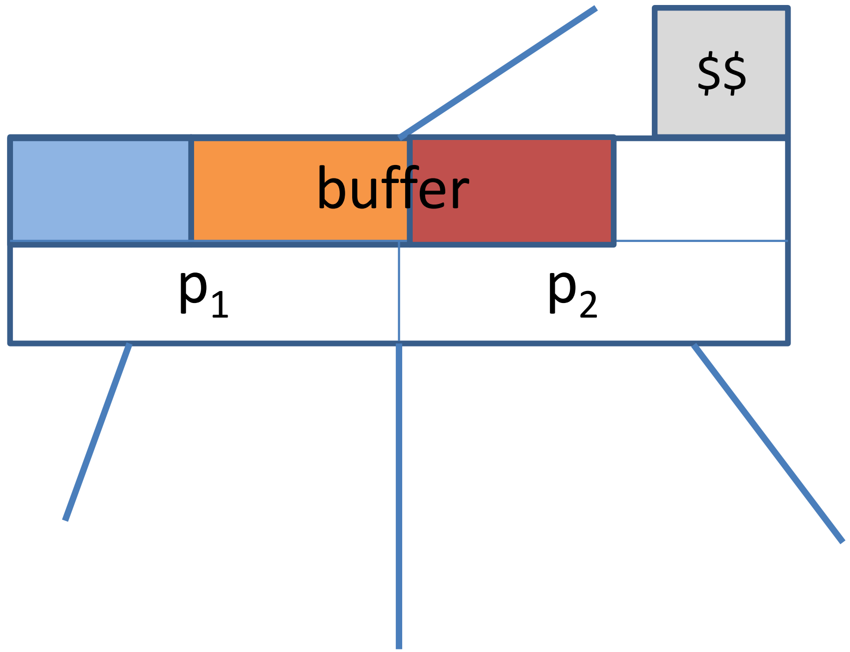
➔ Each item contributes $\Theta(1/B)$ to the bank account is enough!

Buffer Tree

Amortized Analysis

Cost of splitting/merging buffers:

- Each split/merge costs O(1).
- By previous analysis of (a,b)-tree, a split/merge only occurs (at most) once every B-1 operations.
- Thus each operation is charge O(1/B) per split/merge.



Each split/merge costs O(1).

A split/merge only occurs when at least B-1 operations occur.

➔ Each item contributes θ(1/B) to the bank account is enough!

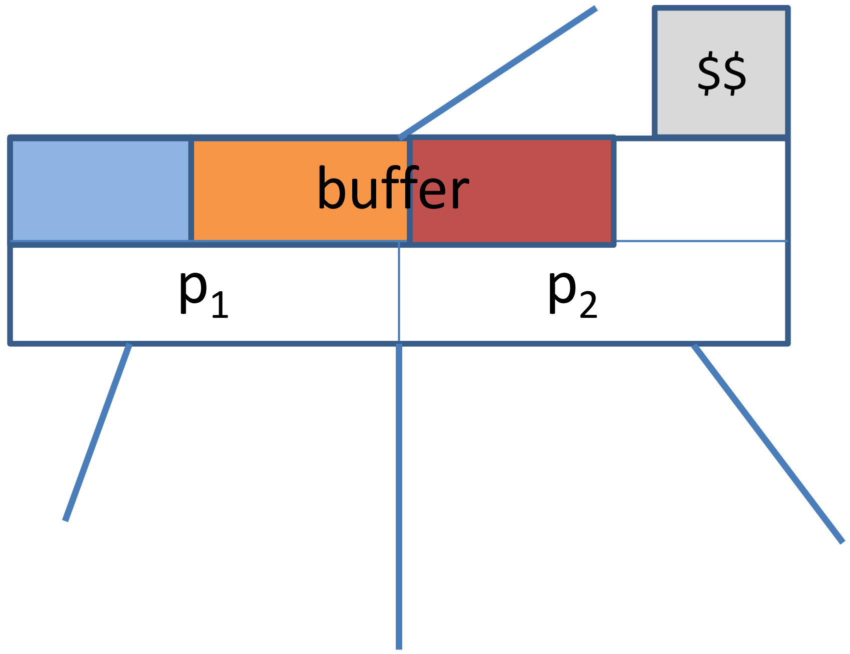
Buffer Tree

Amortized Analysis

Each node has a bank account.

Every operation:

- If root-to-leaf path for a key touches a node, it adds $\theta(1/B)$ dollars to the bank account for that node.



root-to-leaf path: $O(\log n)$

Conclusion:

Cost of insert/delete is: $O\left(\frac{1}{B}\log n\right)$

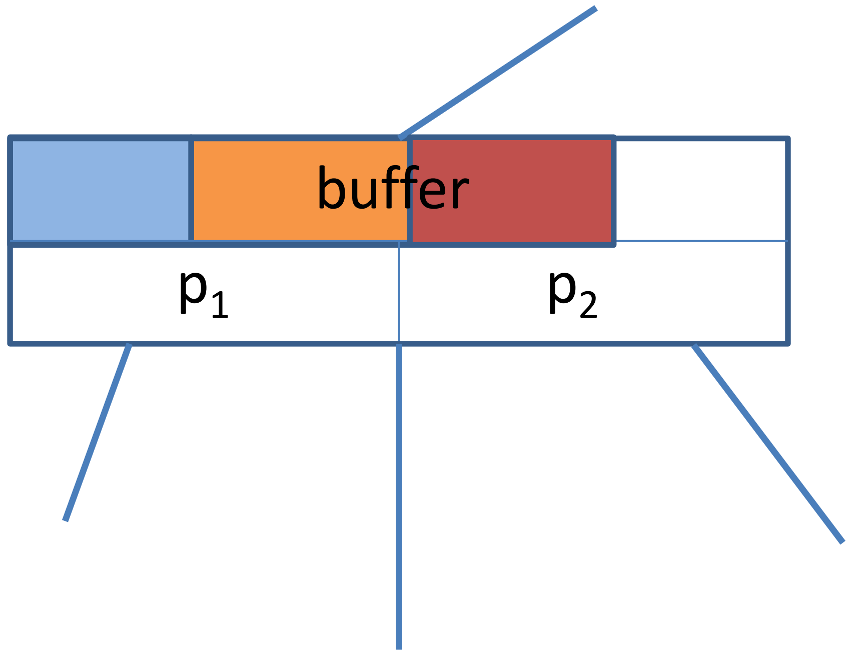
Buffer Tree

Summary

Cost of operations:

insert/delete: $O\left(\frac{1}{B}\log n\right)$

search: $O\left(\log n\right)$



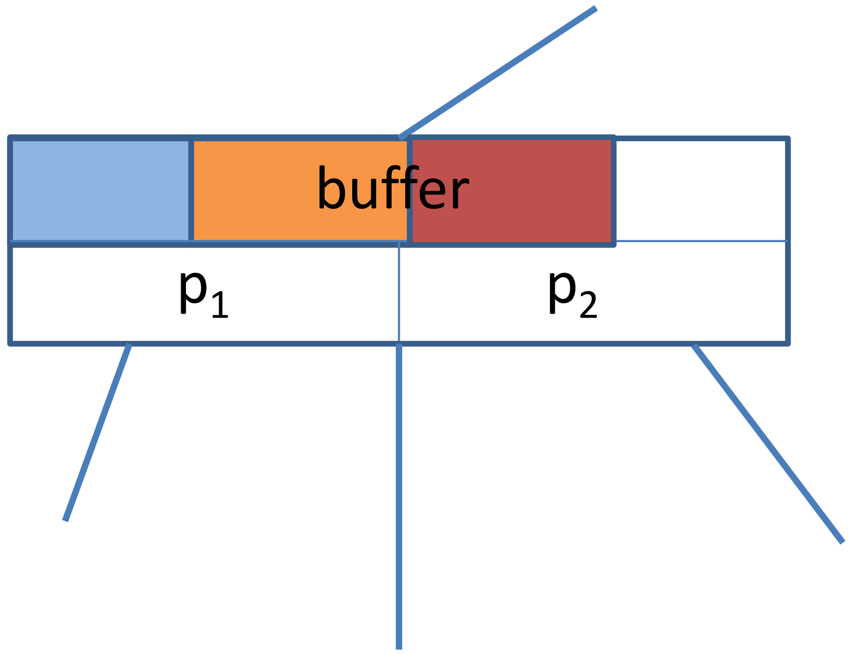
Buffer Tree

Better trade-off:

What if degree of each node is increased to: $\sqrt{B}$

insert/delete: ??

search: ??



$$(\sqrt{B}, 2\sqrt{B}) - \text{tree}$$

Buffer Tree

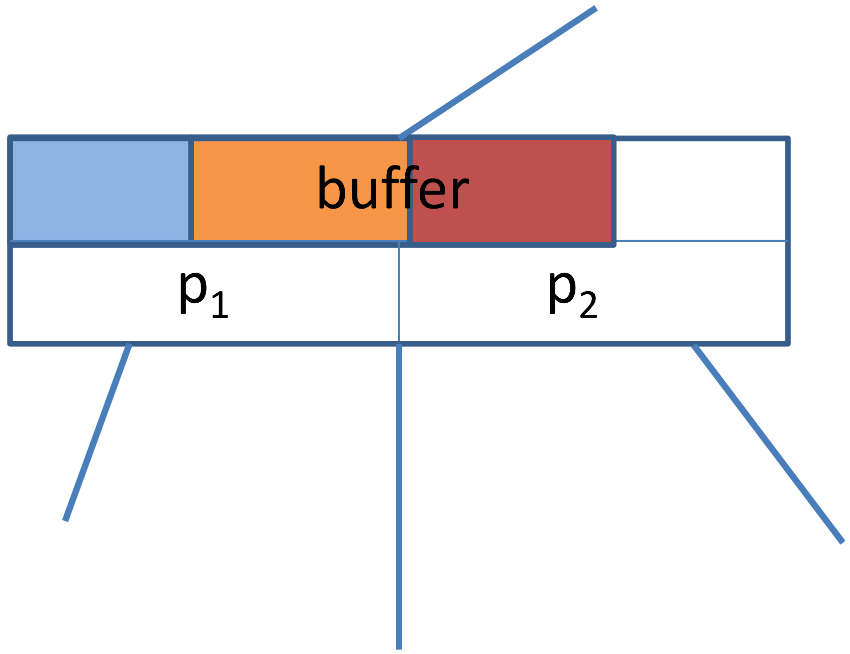
Better trade-off:

What if degree of each node is increased to: $\sqrt{B}$

What if degree of each node is increased to: B^{ϵ}

insert/delete: ??

search: ??



$(B^{\epsilon}, 2B^{\epsilon})$-tree

Today's Plan

Searching and Sorting

1. B-trees
   - ⇒ Algorithm
   - ⇒ Amortized analysis

2. Buffer trees
   - ⇒ Write-optimized data structures
   - ⇒ Buffered data structures
   - ⇒ Amortized analysis

3. van Emde Boas Search Tree
   - ⇒ Cache-oblivious algorithms
   - ⇒ van Emde Boas memory layout

Cache Oblivious Search Trees

What if you do not know the value of B or M?

Cache size differ on every machine, on every architecture, and at different levels of the caching hierarchy. Without knowing the specific hardware, how do you optimize properly?

Cache Oblivious Search Trees

Idea:

Design an algorithm that does not know B or M.

Analyze the algorithm in the external memory model (where B and M are known).

Cache Oblivious Search Trees

Example: an array

An algorithm for scanning an array from beginning to end does not depend on B or M.

The running time for an algorithm to scan an array of size n is O(n/B).

Cache Oblivious Search Trees

Today:

<u>Static</u> cache-oblivious search tree.

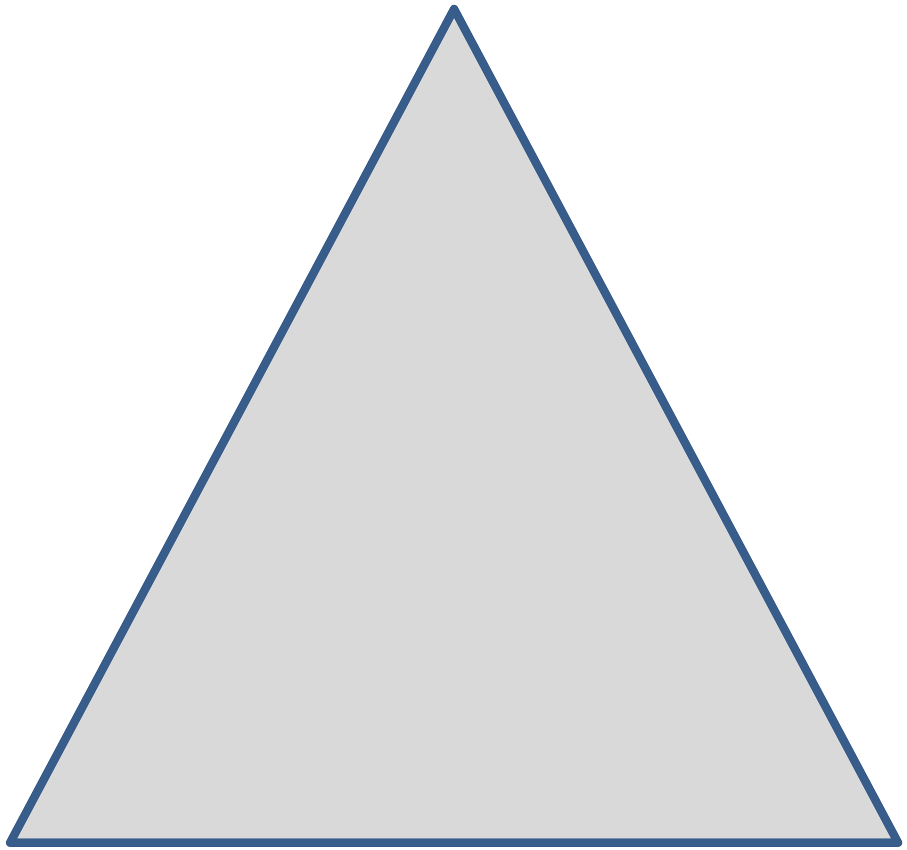
Goal: build a tree that supports efficient search operations.

(We will not support insert and delete. See research papers.)

Recursive Memory Layout: van Emde Boas

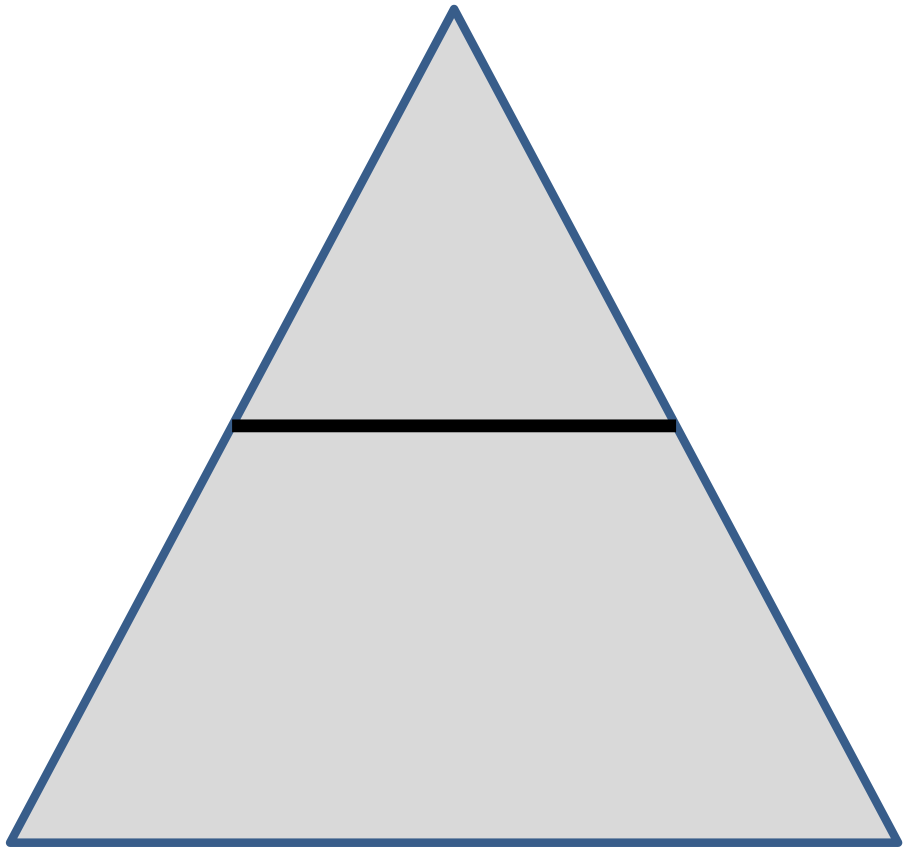
Recursive Memory Layout: van Emde Boas

1. Start with a (perfectly) balanced binary search tree.



Recursive Memory Layout: van Emde Boas

1. Start with a (perfectly) balanced binary search tree.
2. Divide it in half, from top to bottom.

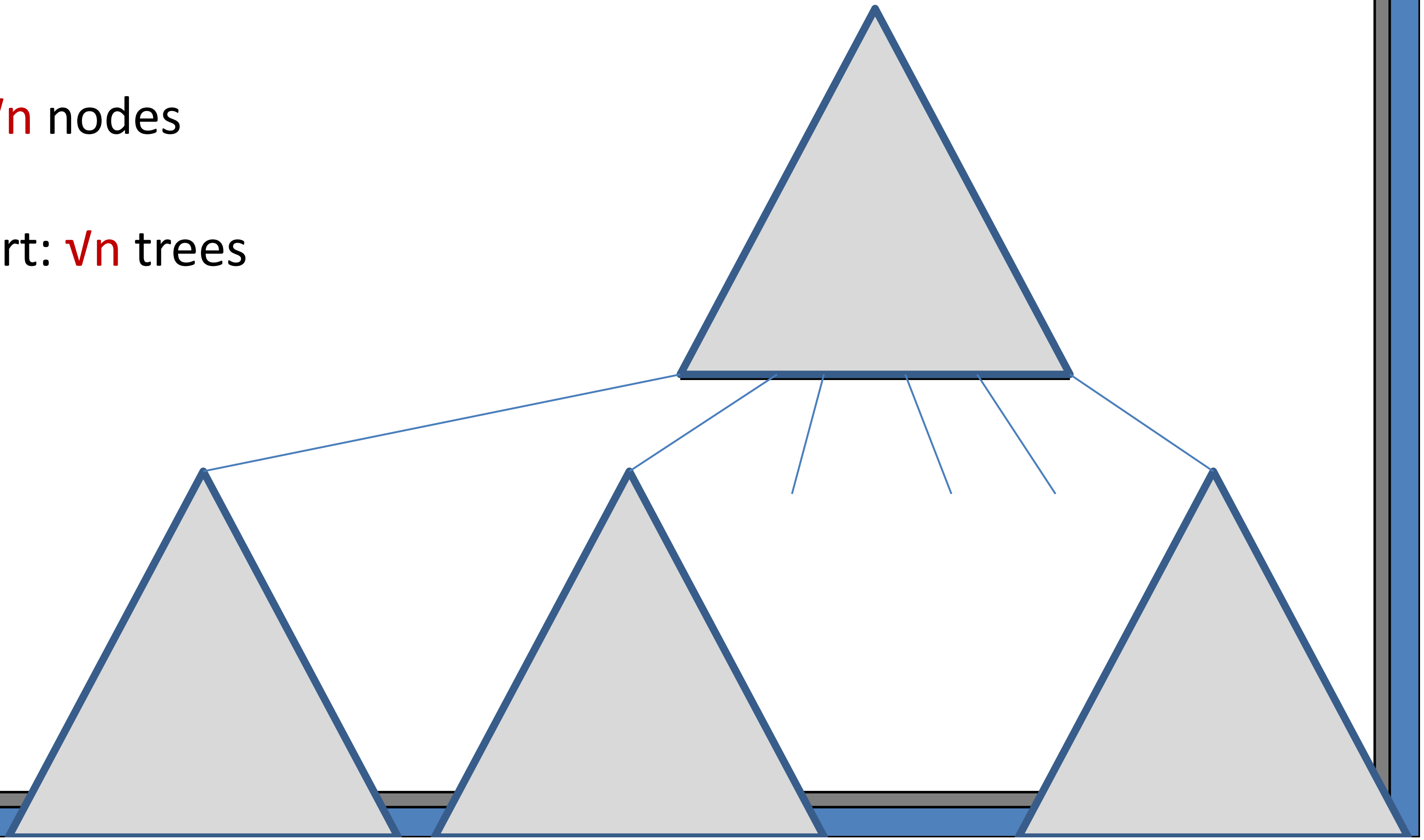


Recursive Memory Layout: van Emde Boas

1. Start with a (perfectly) balanced binary search tree.
2. Divide it in half, from top to bottom.

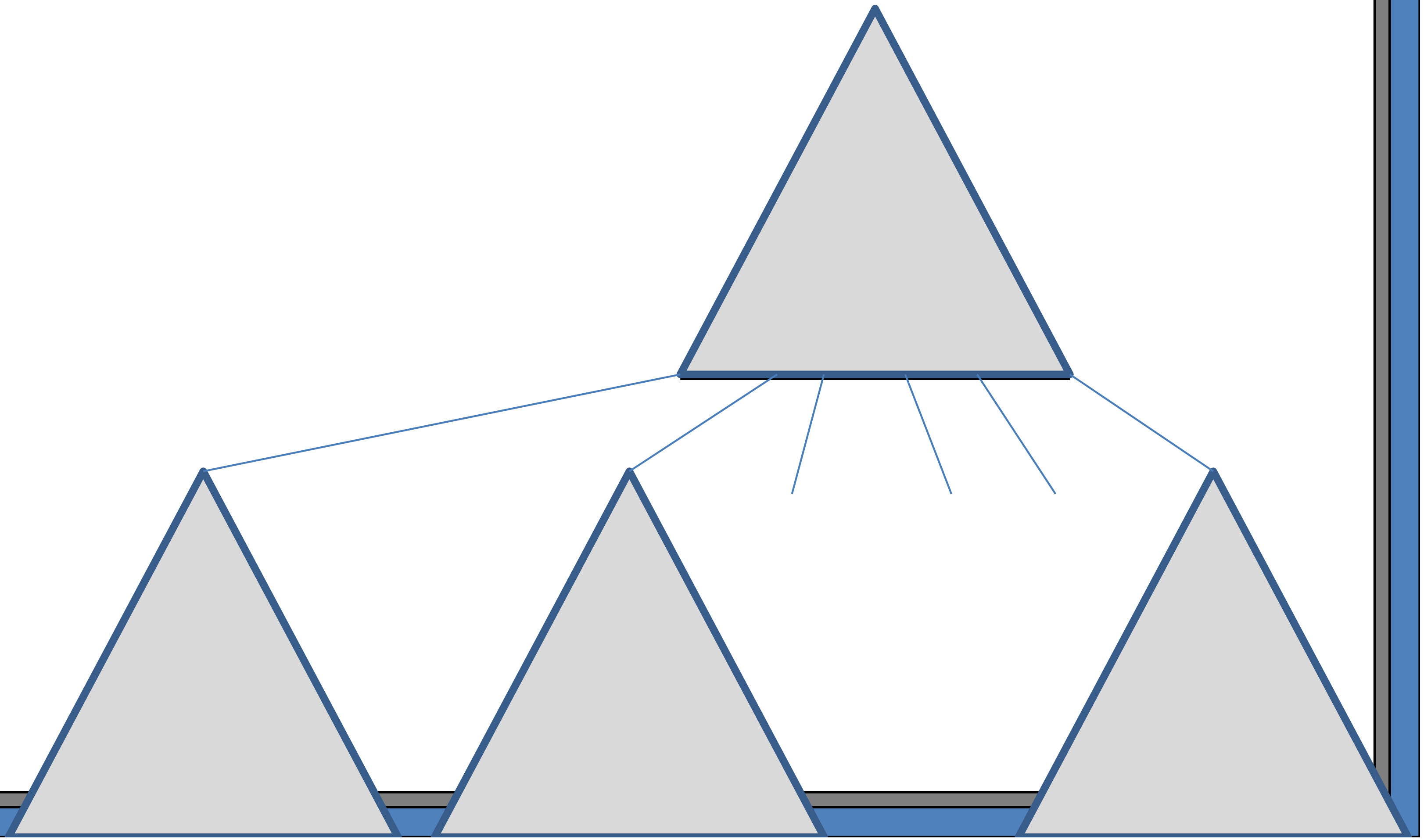
Top part: √n nodes

Bottom part: √n trees



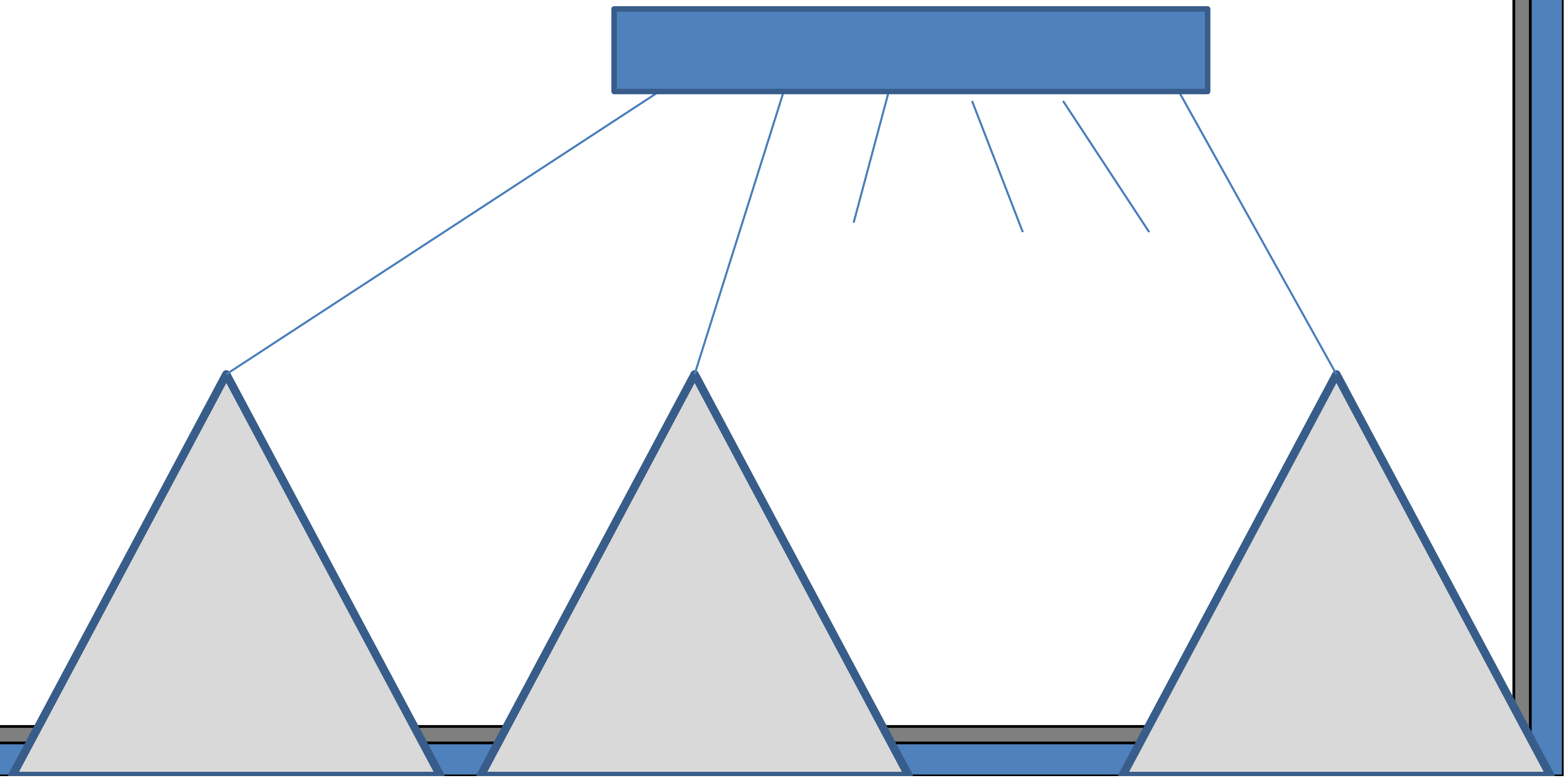
Recursive Memory Layout: van Emde Boas

1. Start with a (perfectly) balanced binary search tree.
2. Divide it in half, from top to bottom.
3. Recursively layout each of the √n + 1 trees.



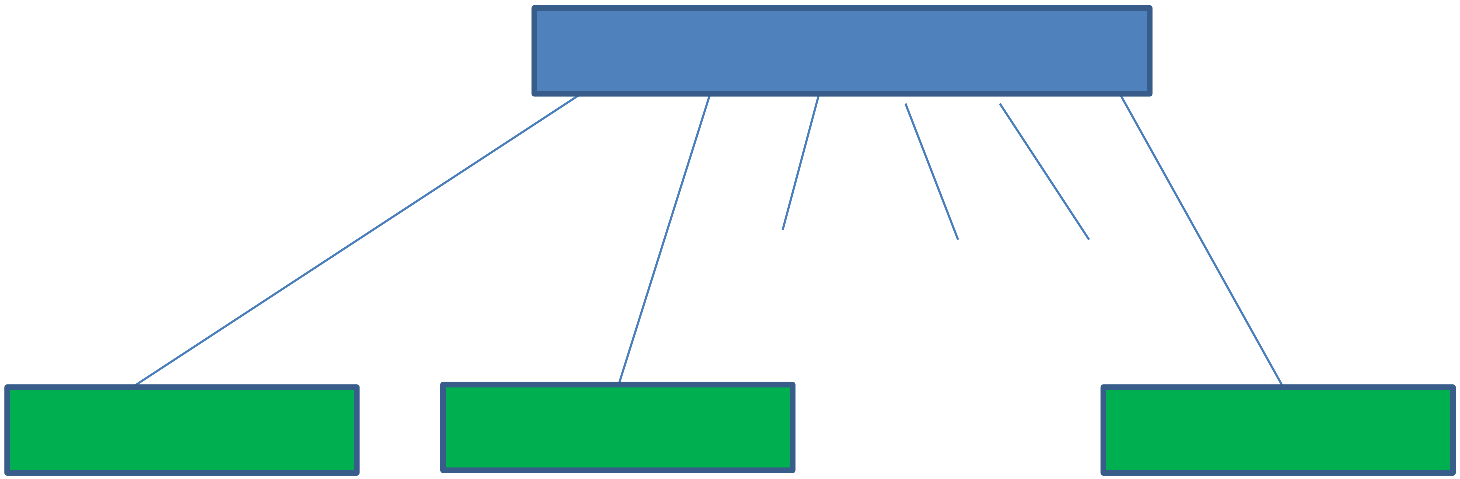
Recursive Memory Layout: van Emde Boas

1. Start with a (perfectly) balanced binary search tree.
2. Divide it in half, from top to bottom.
3. Recursively layout each of the $\sqrt{n} + 1$ trees.



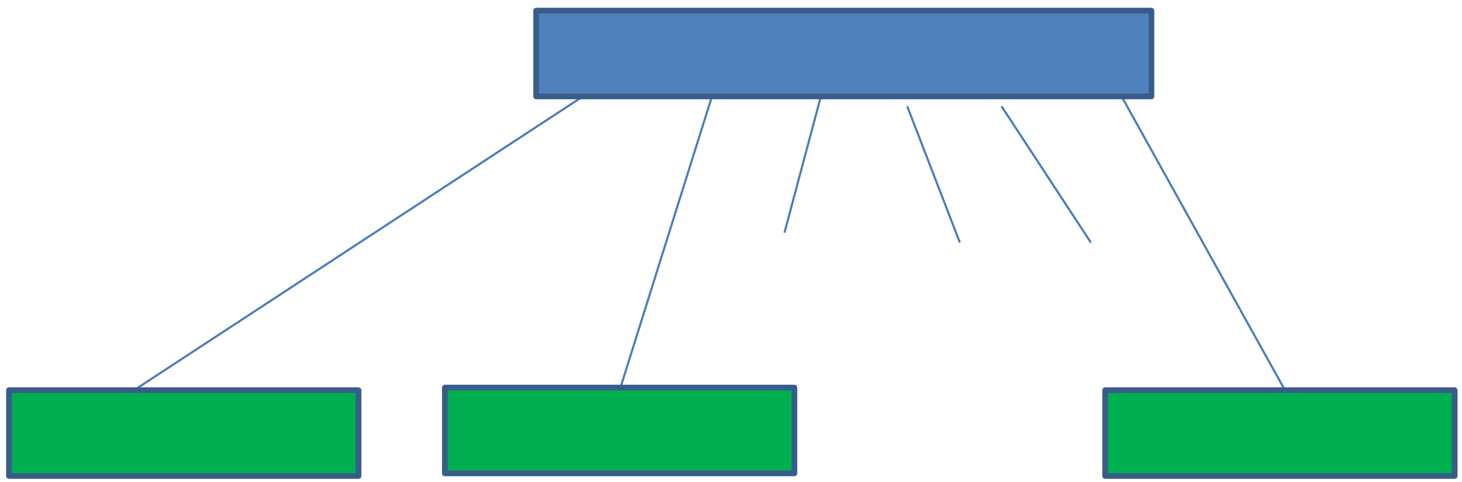
Recursive Memory Layout: van Emde Boas

1. Start with a (perfectly) balanced binary search tree.
2. Divide it in half, from top to bottom.
3. Recursively layout each of the $\sqrt{n} + 1$ trees.



Recursive Memory Layout: van Emde Boas

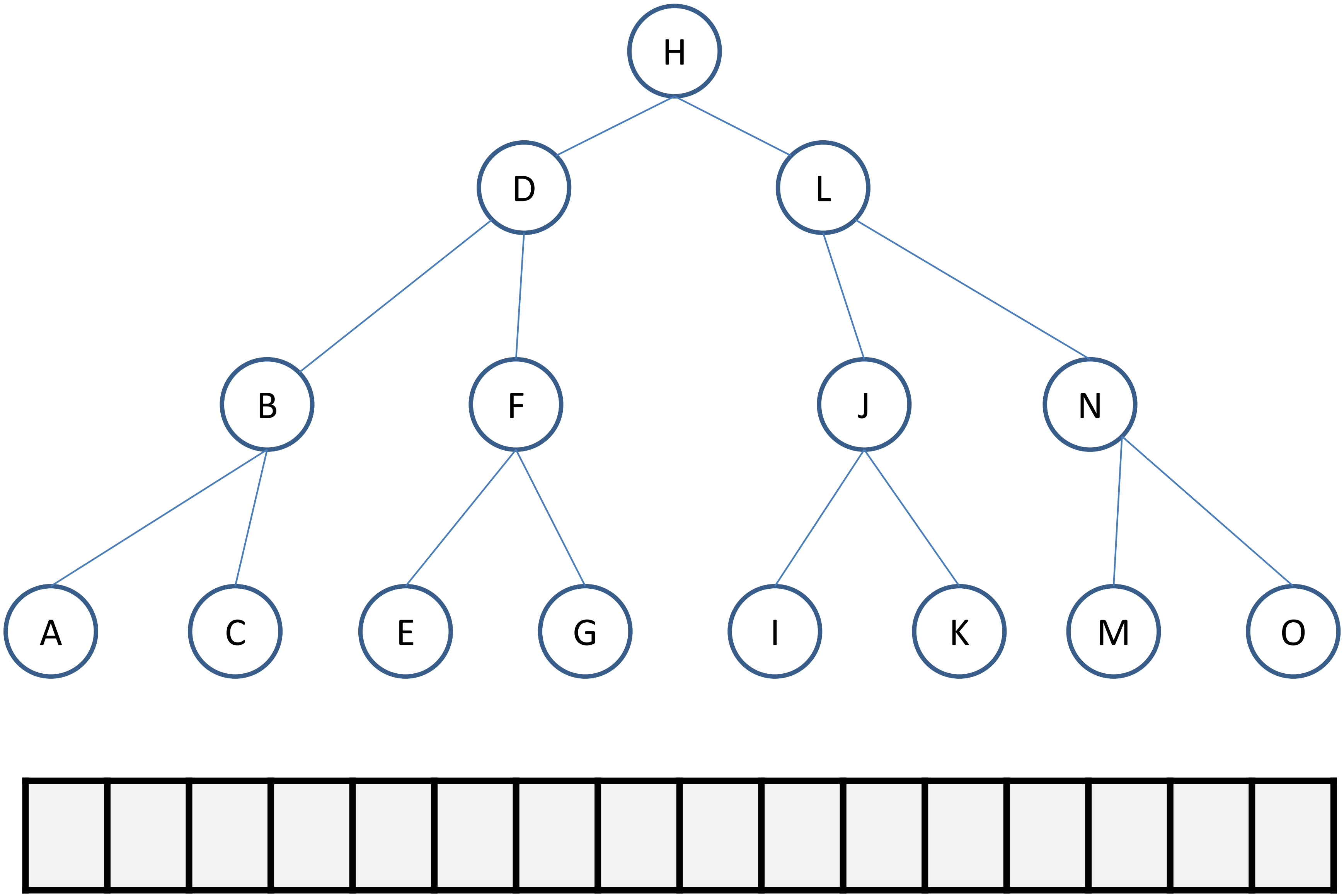
1. Start with a (perfectly) balanced binary search tree.
2. Divide it in half, from top to bottom.
3. Recursively layout each of the $\sqrt{n} + 1$ trees.
4. Layout: root, followed by children in order.



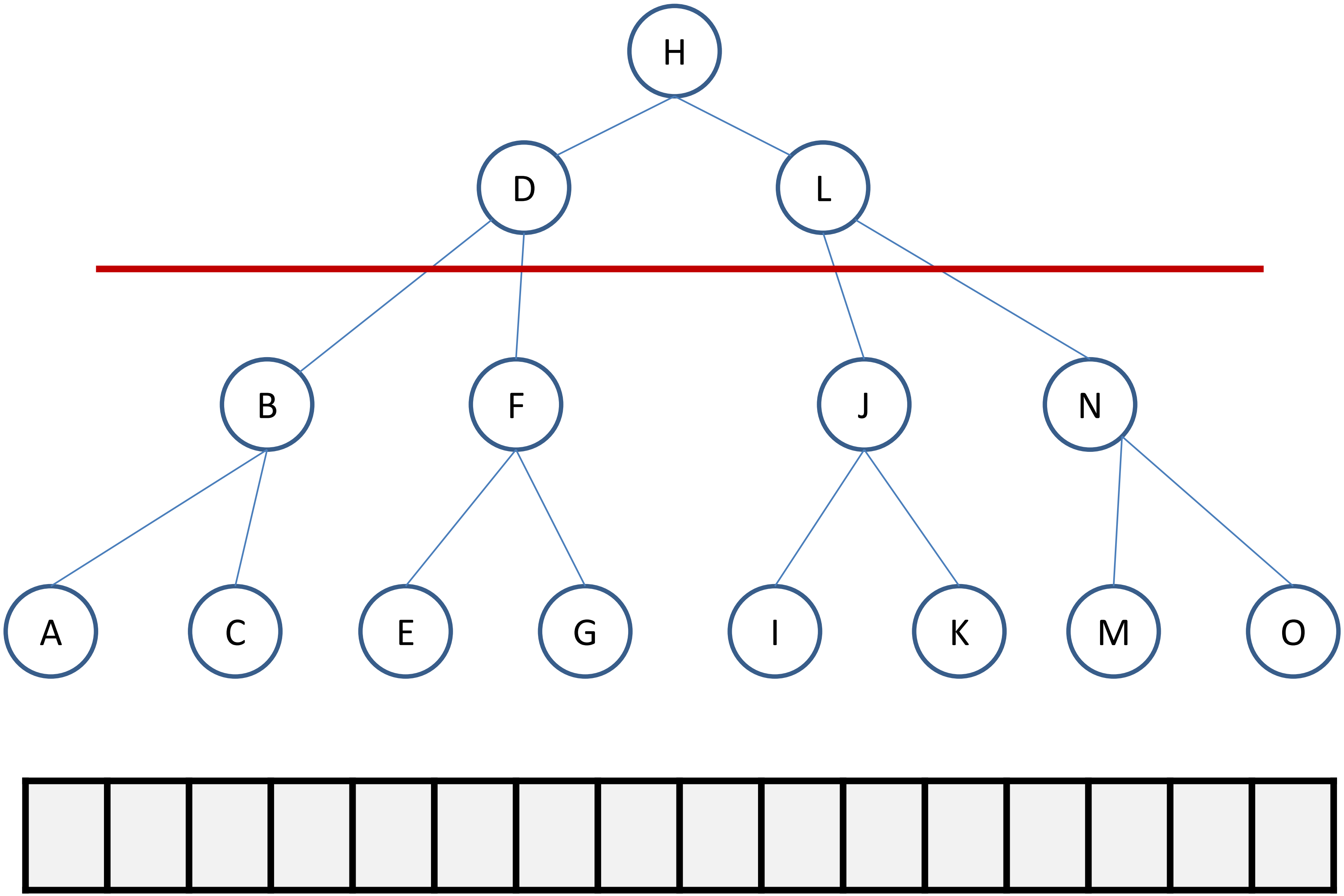
Recursive Memory Layout: van Emde Boas

1. Start with a (perfectly) balanced binary search tree.
2. Divide it in half, from top to bottom.
3. Recursively layout each of the $\sqrt{n} + 1$ trees.
4. Layout: root, followed by children in order.

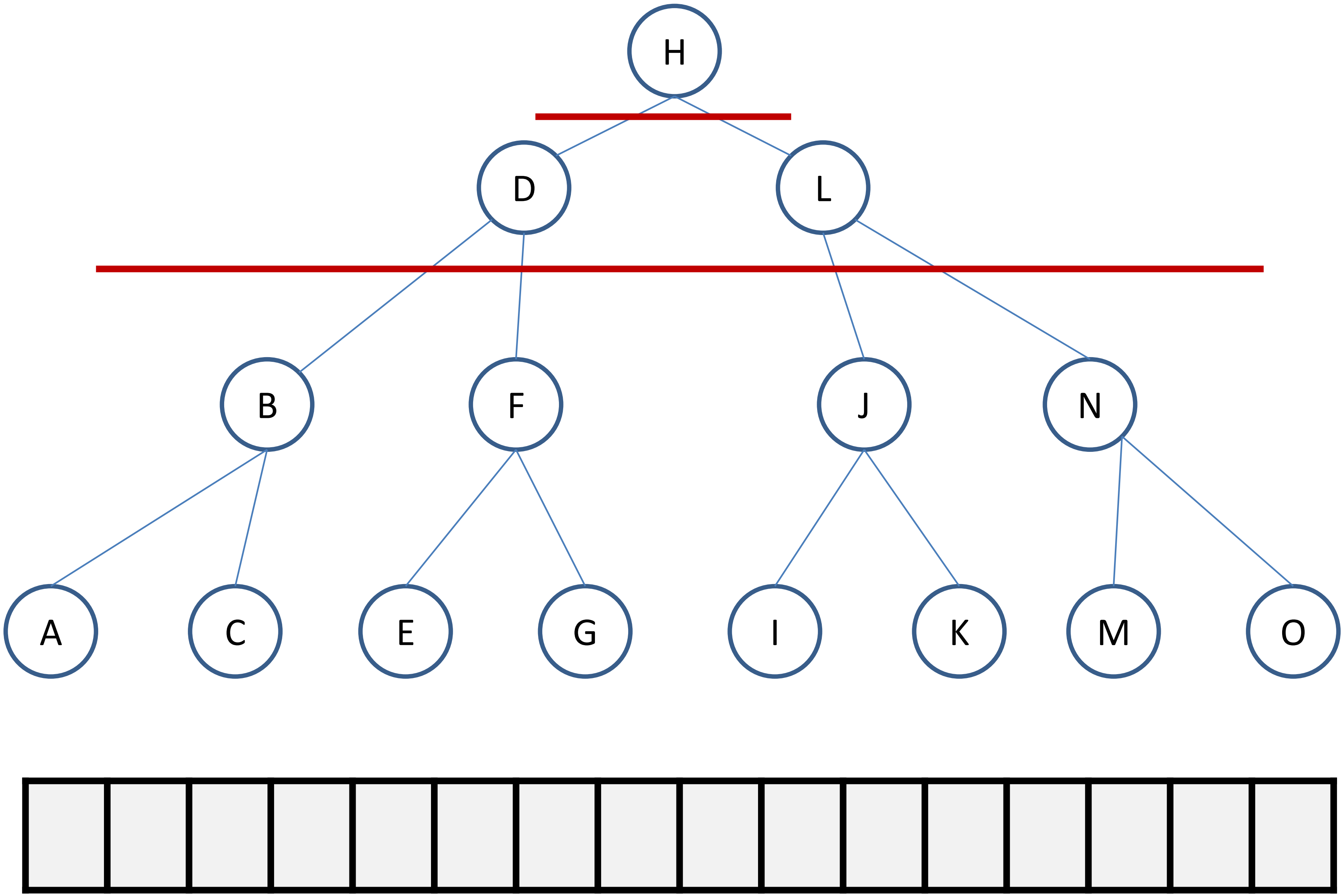
Example



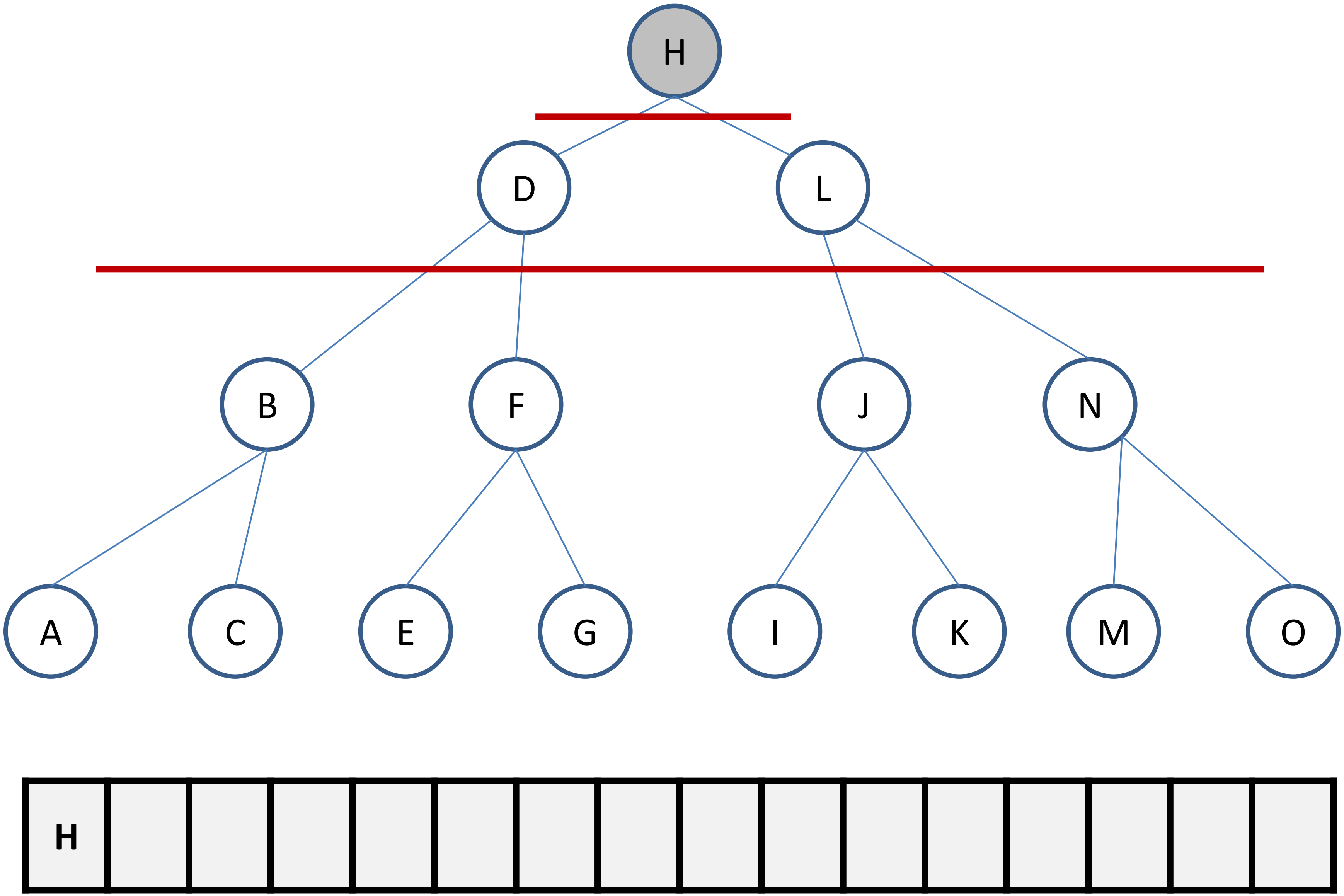
Example



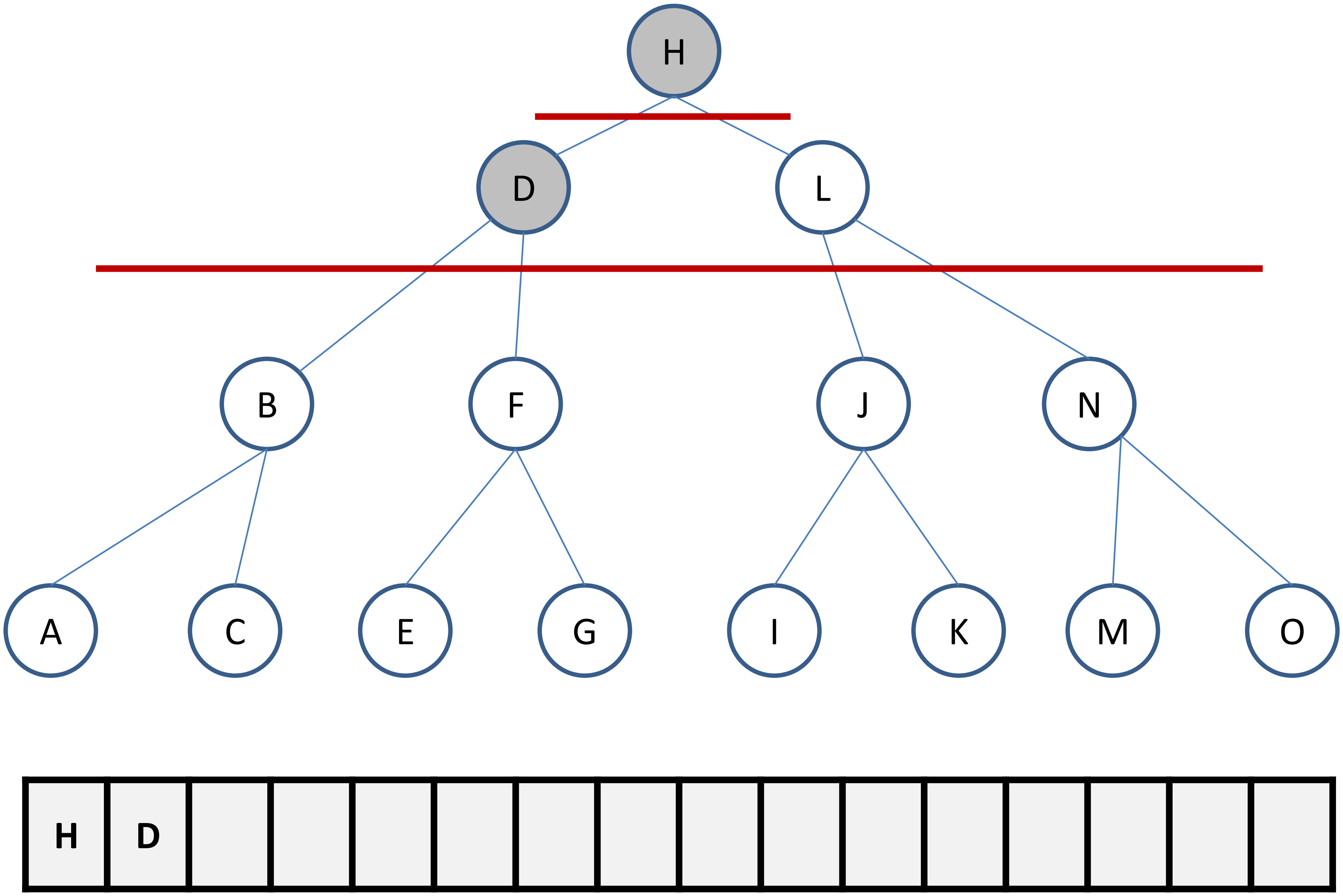
Example



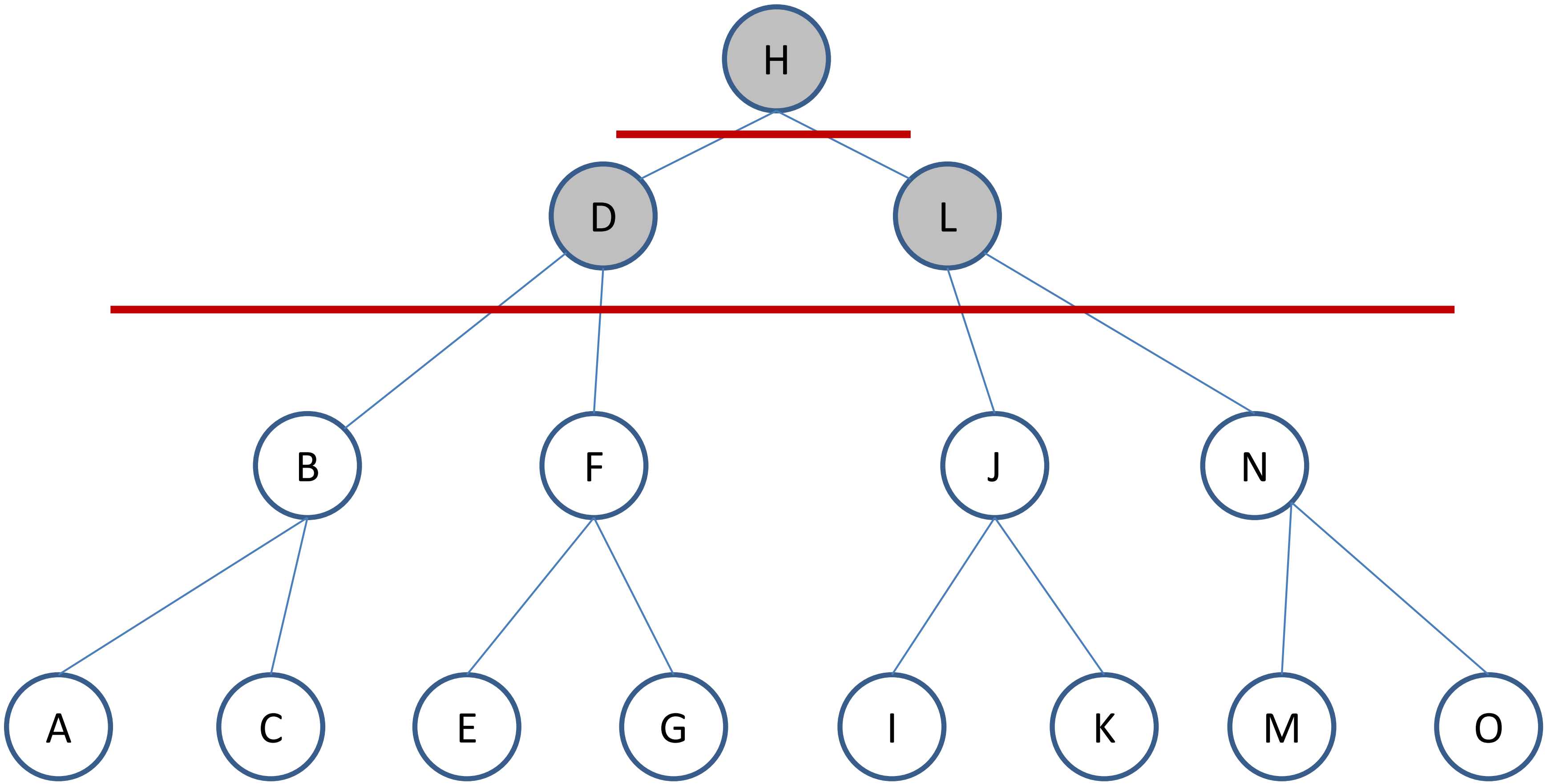
Example



Example

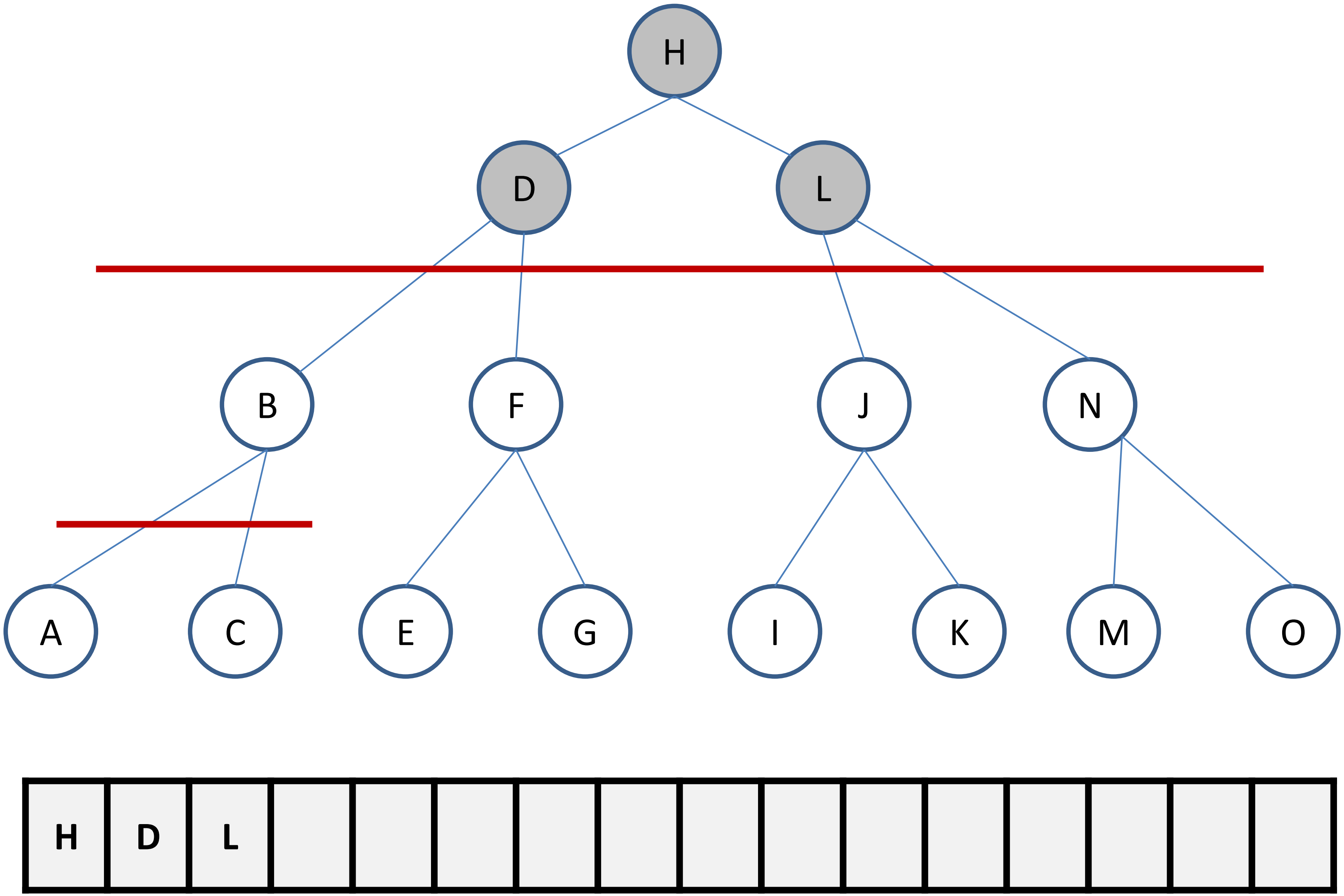


Example

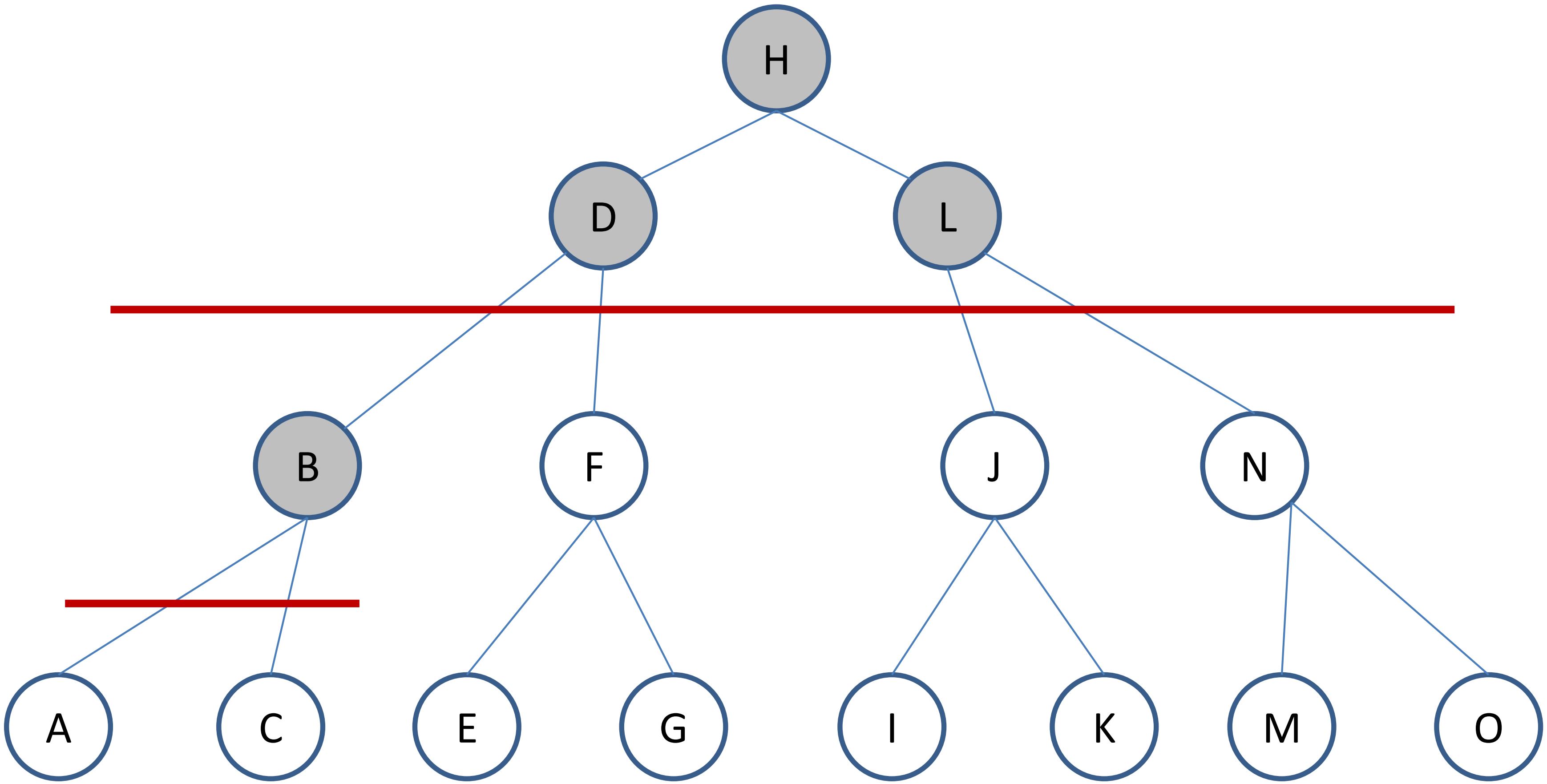


H	D	L													

Example

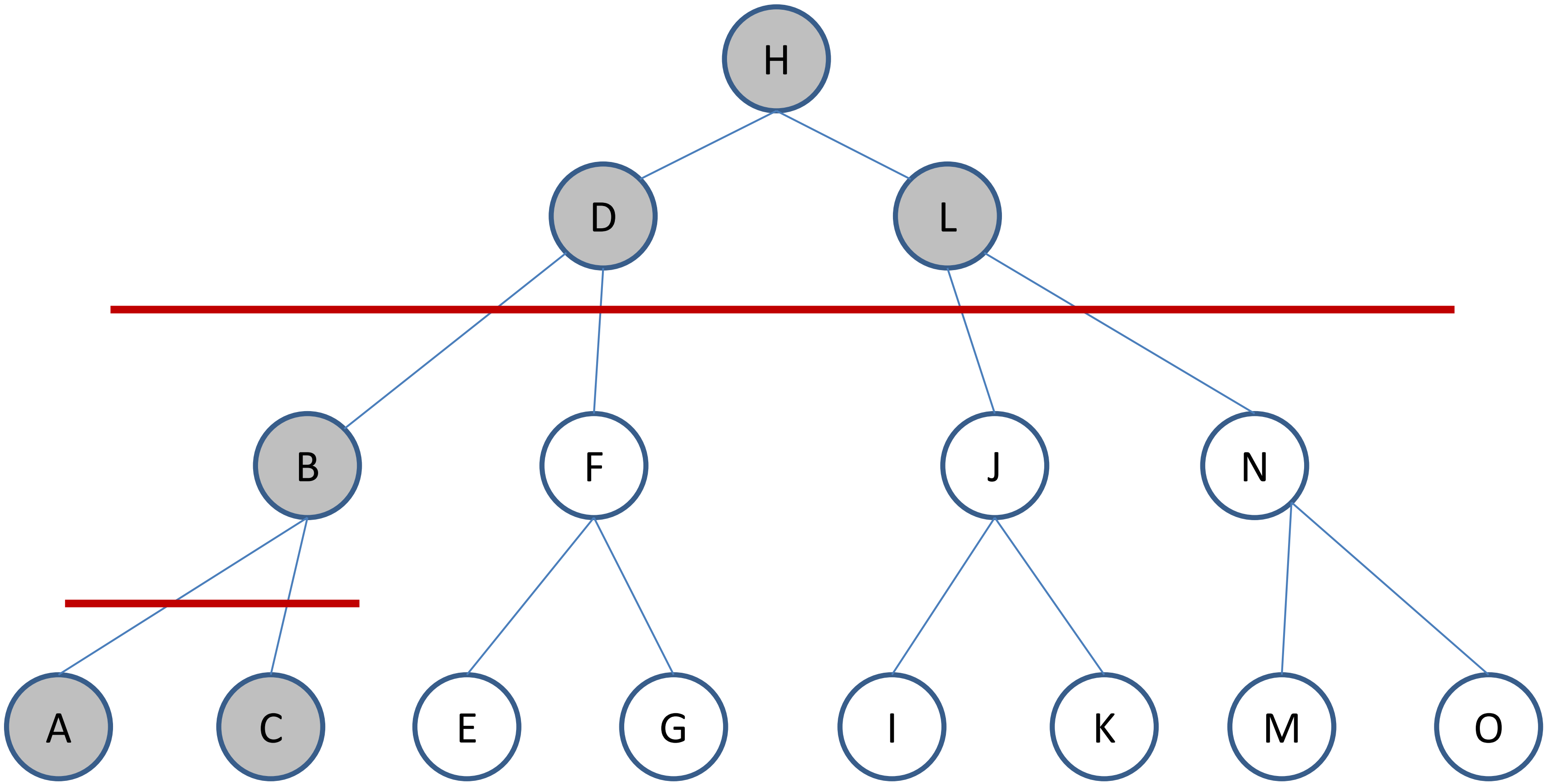


Example



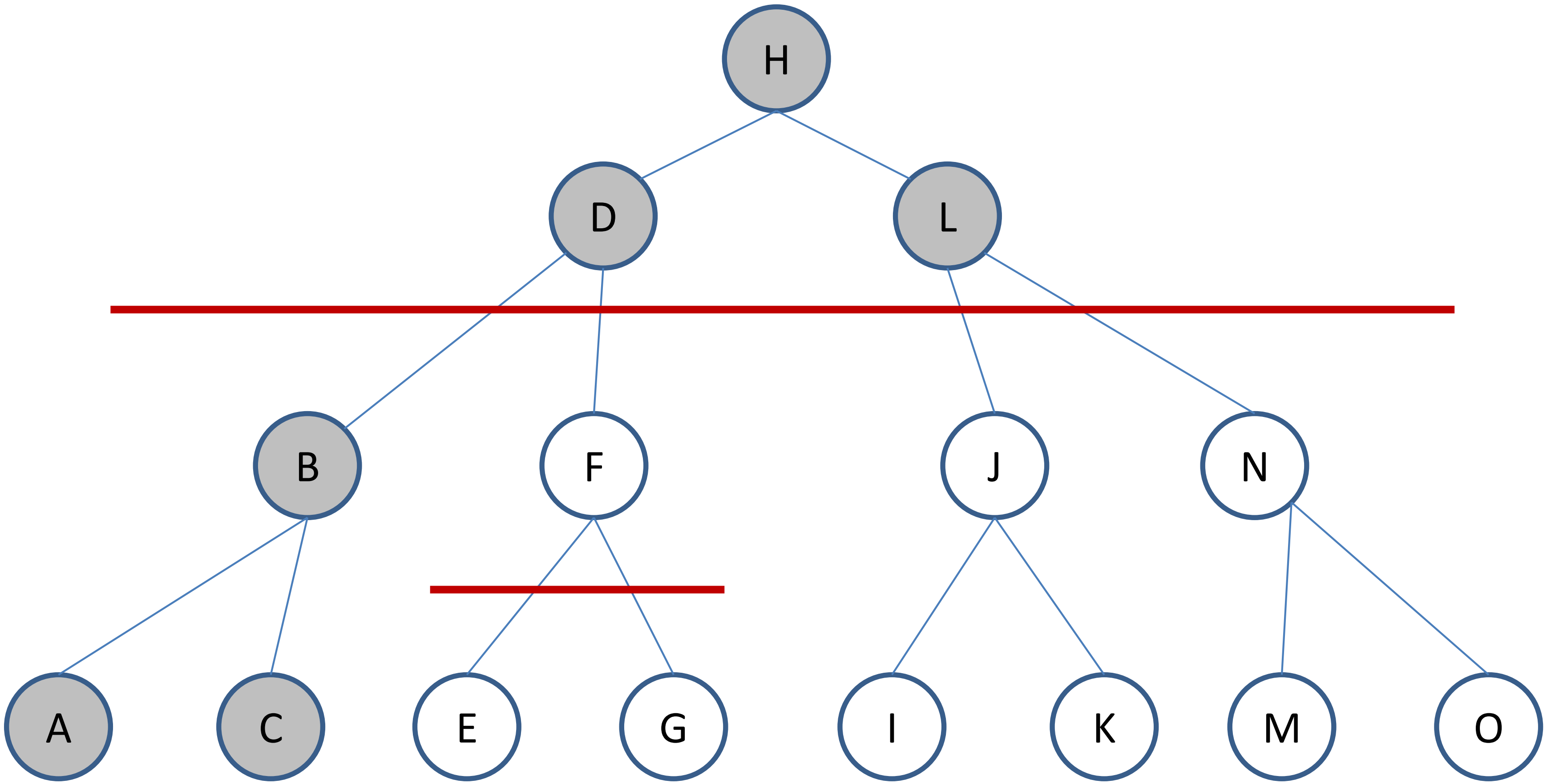
H	**D**	**L**	**B**												

Example



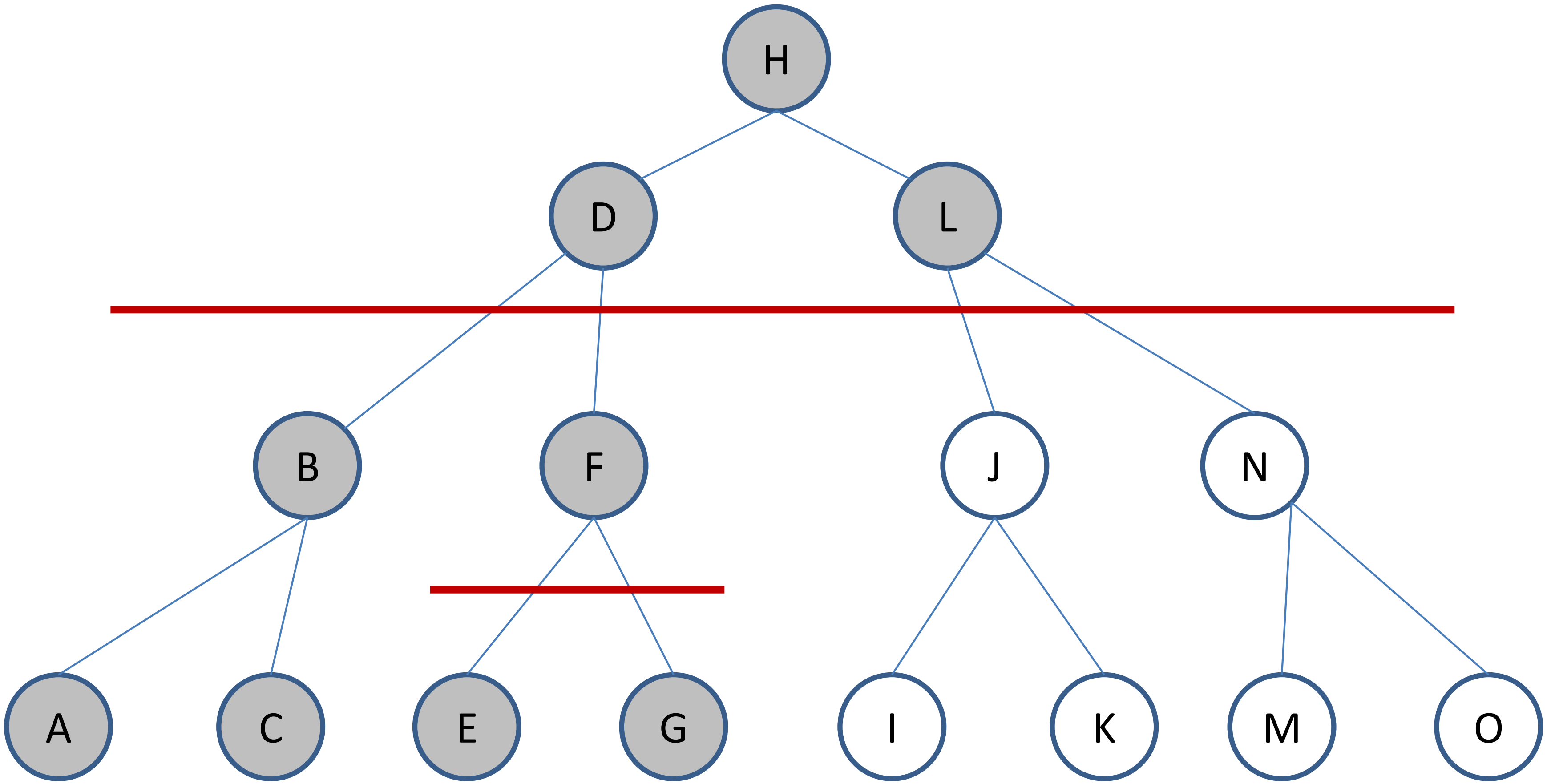
H	**D**	**L**	**B**	**A**	**C**										

Example



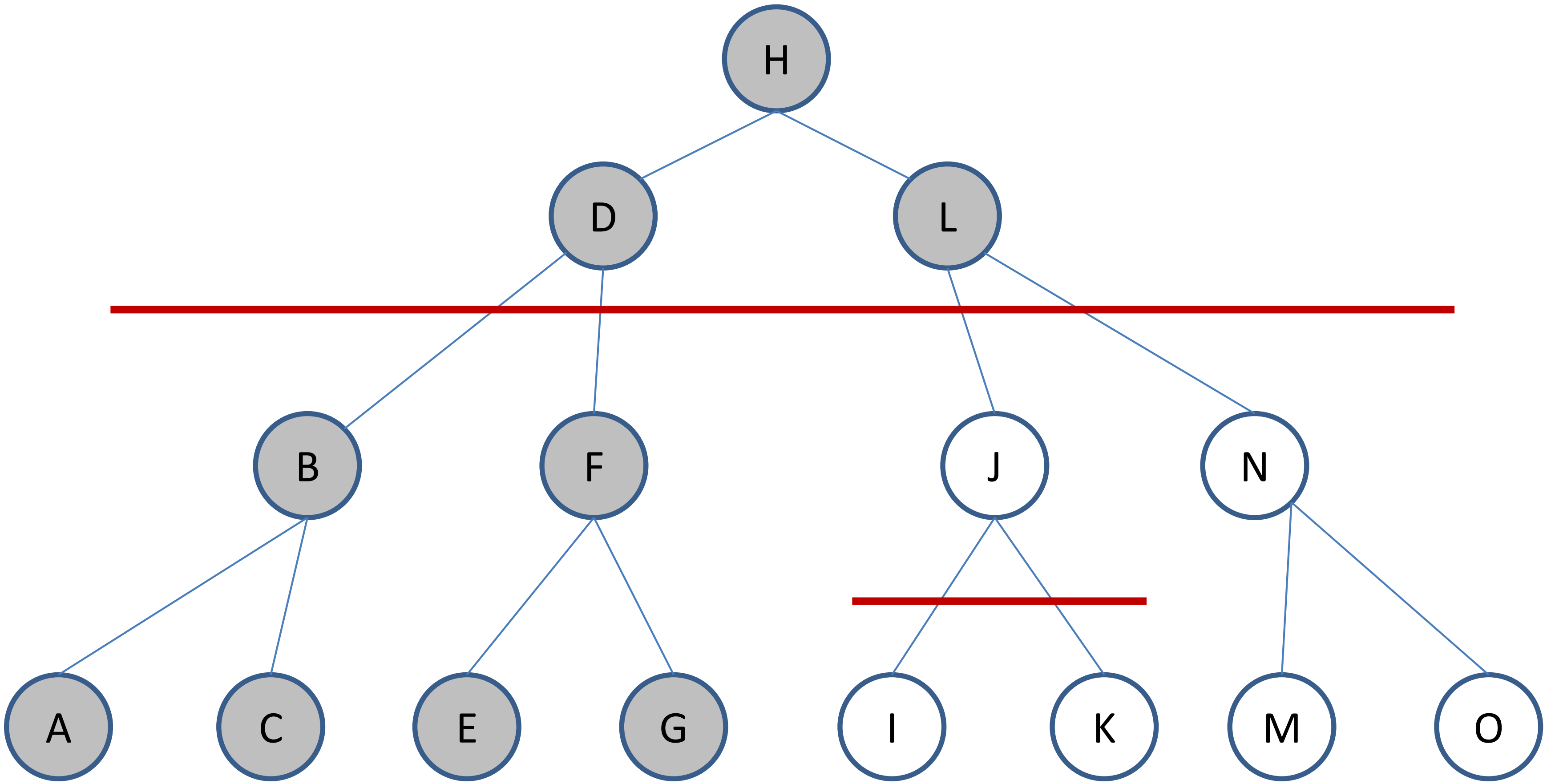
H	**D**	**L**	**B**	**A**	**C**										

Example



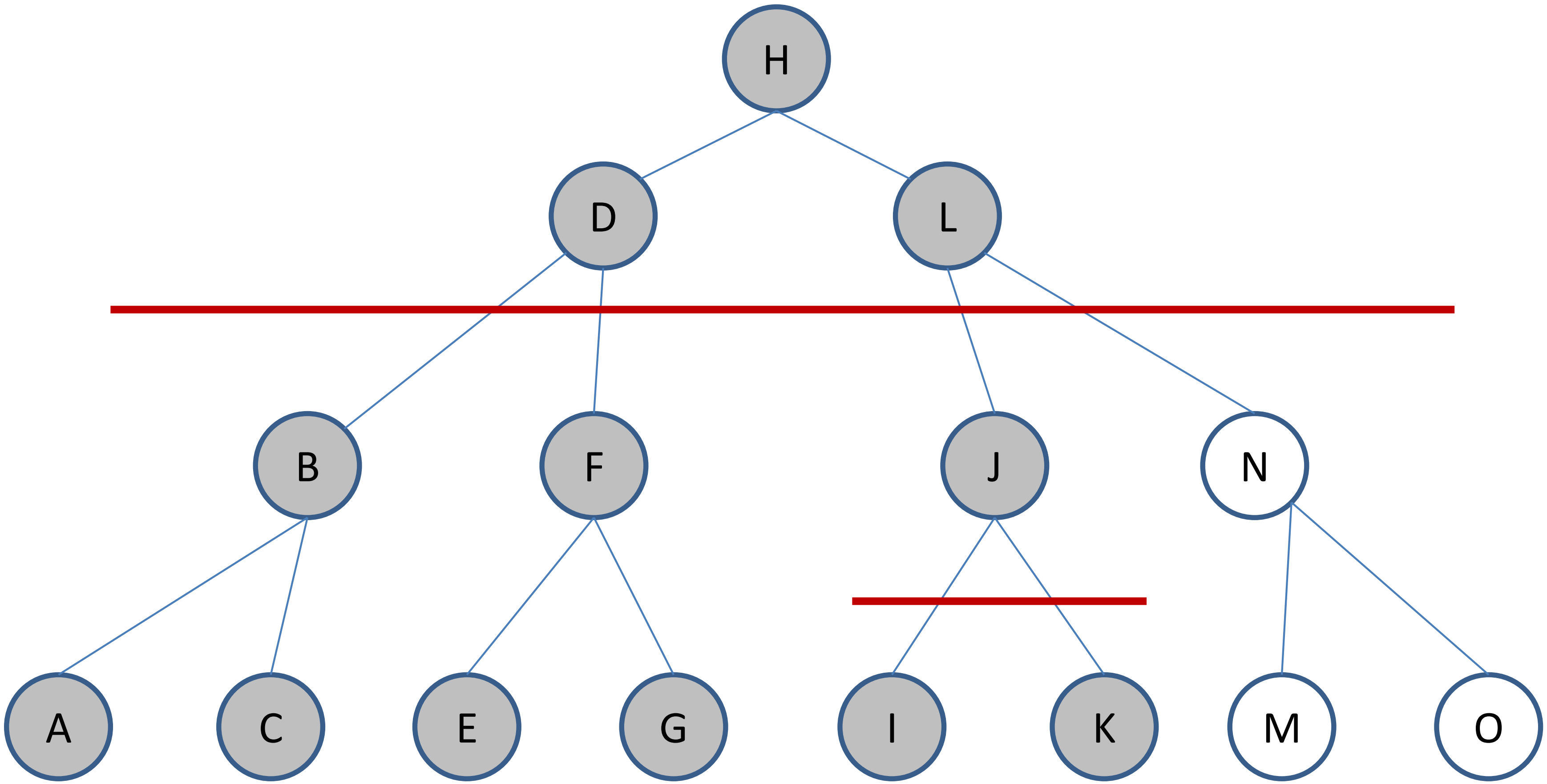
H	**D**	**L**	**B**	**A**	**C**	**F**	**E**	**G**							

Example



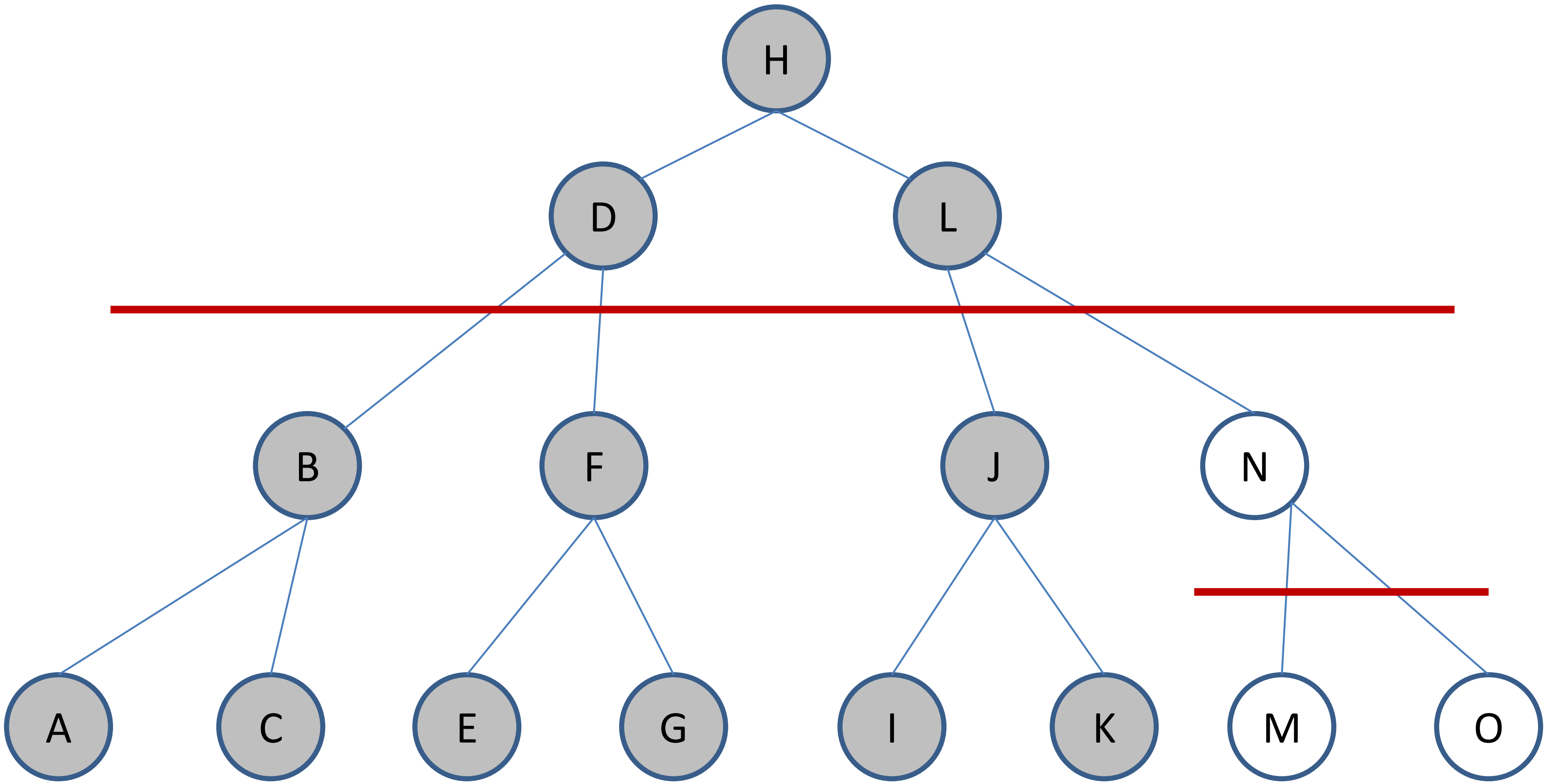
H	**D**	**L**	**B**	**A**	**C**	**F**	**E**	**G**							

Example



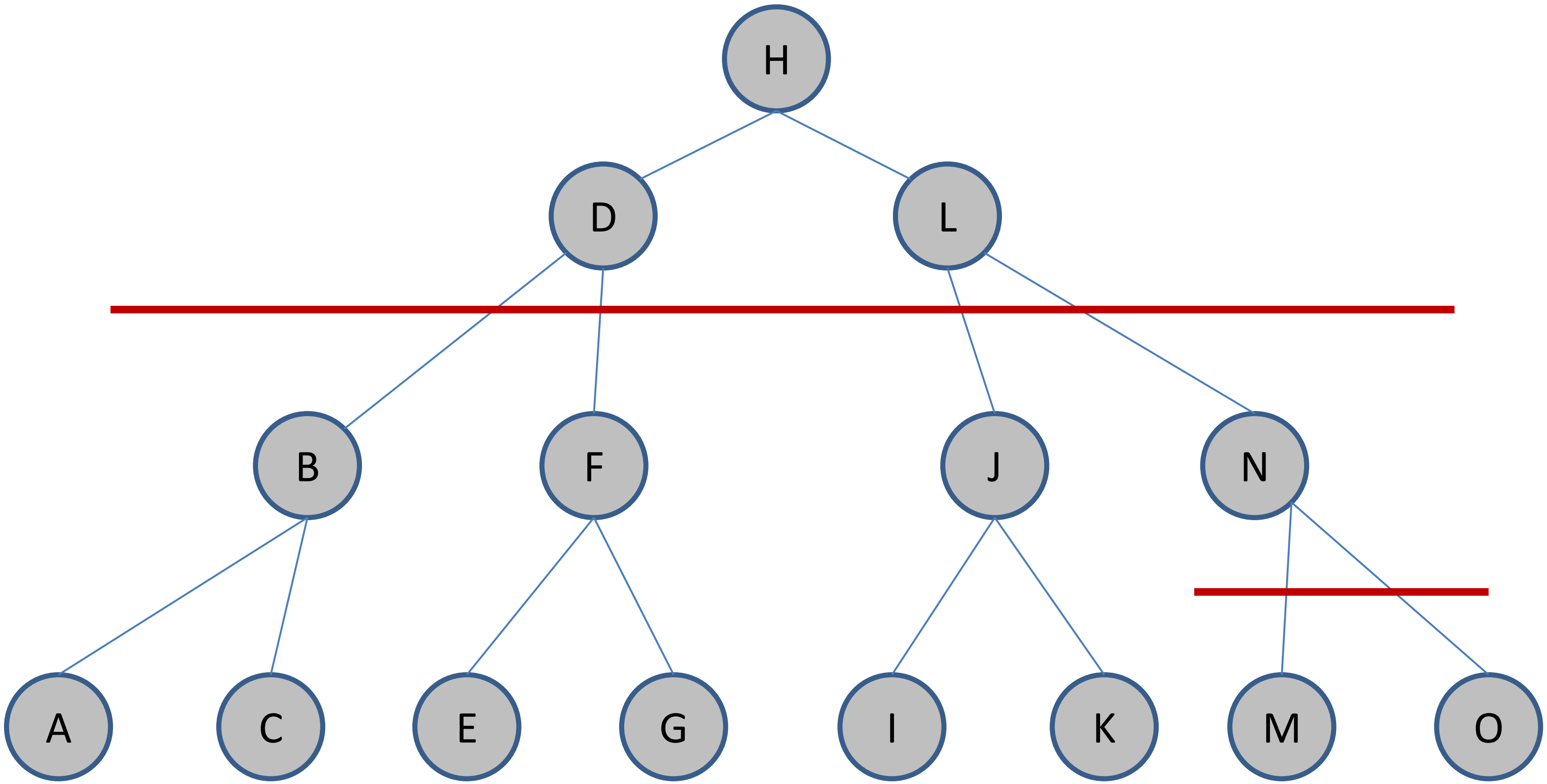
H	**D**	**L**	**B**	**A**	**C**	**F**	**E**	**G**	**J**	**I**	**K**				

Example



H	**D**	**L**	**B**	**A**	**C**	**F**	**E**	**G**	**J**	**I**	**K**				

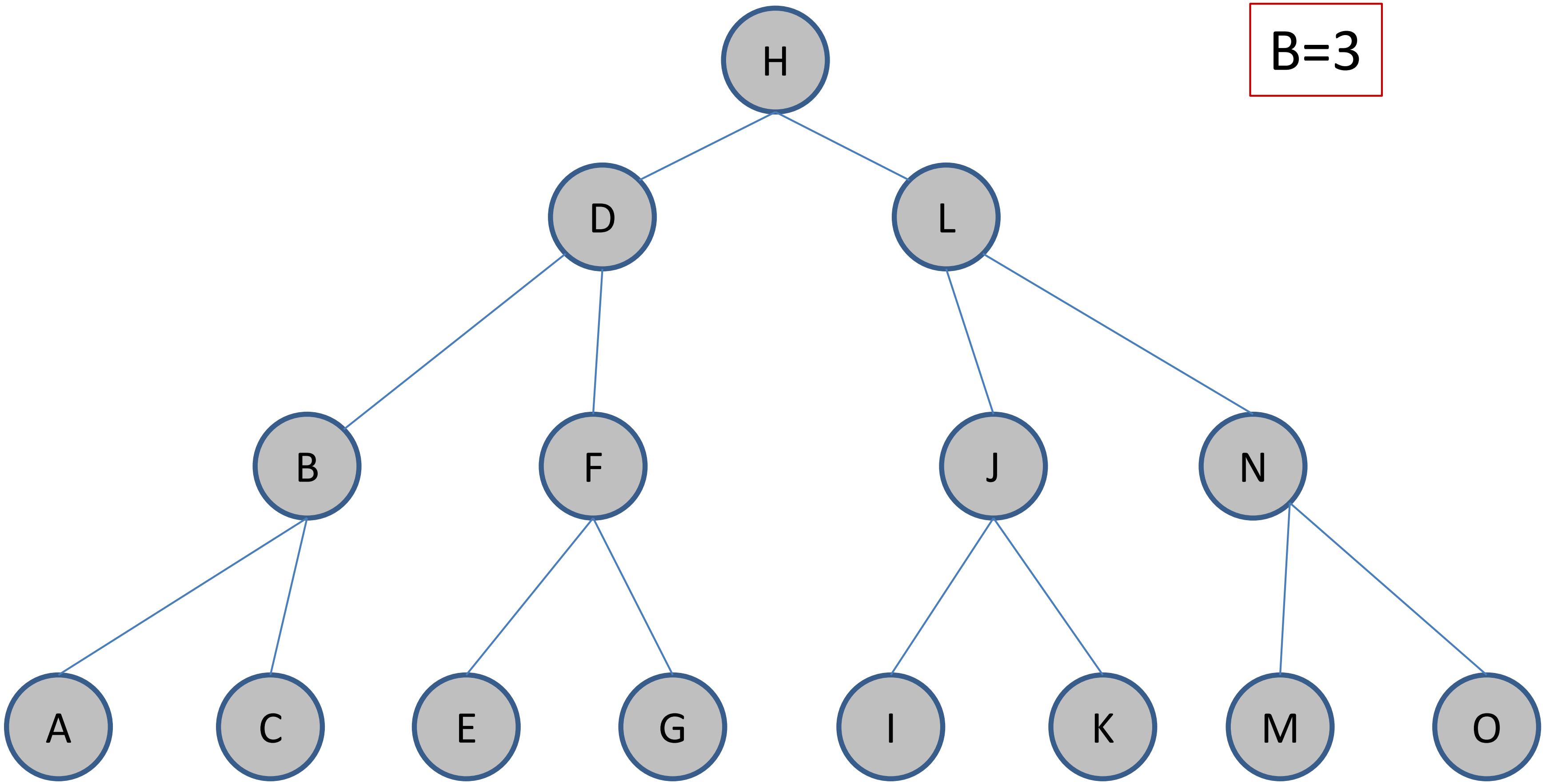
Example



H	**D**	**L**	**B**	**A**	**C**	**F**	**E**	**G**	**J**	**I**	**K**	**N**	**M**	**O**	

Search Operation

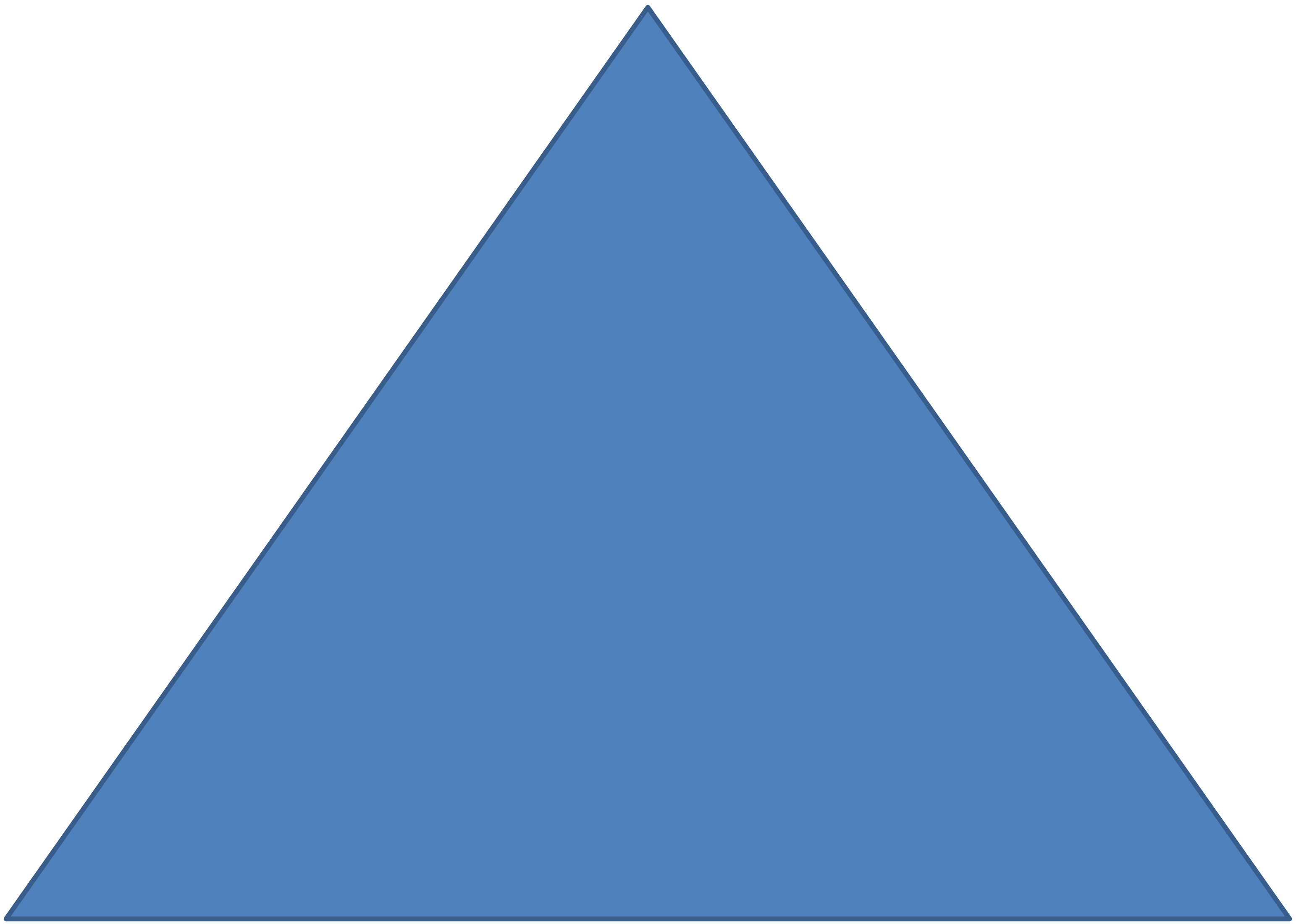
B=3



H	D	L	B	A	C	F	E	G	J	I	K	N	M	O	

van Emde Boas Layout: Analysis

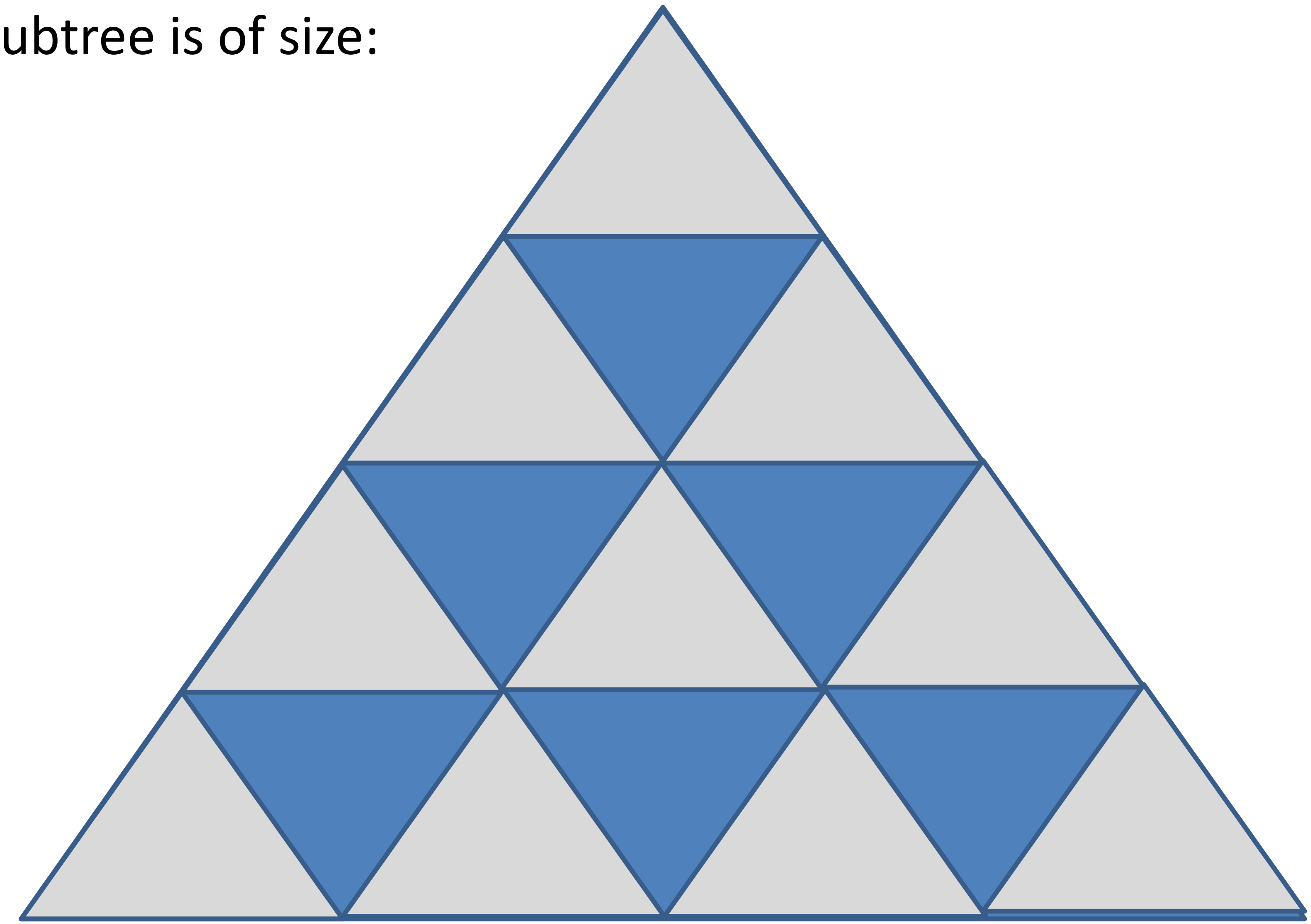
Start with the balanced binary tree.



van Emde Boas Layout: Analysis

Run the recursive decomposition until each subtree is of size:

- $> \sqrt{B}$
- $< B$



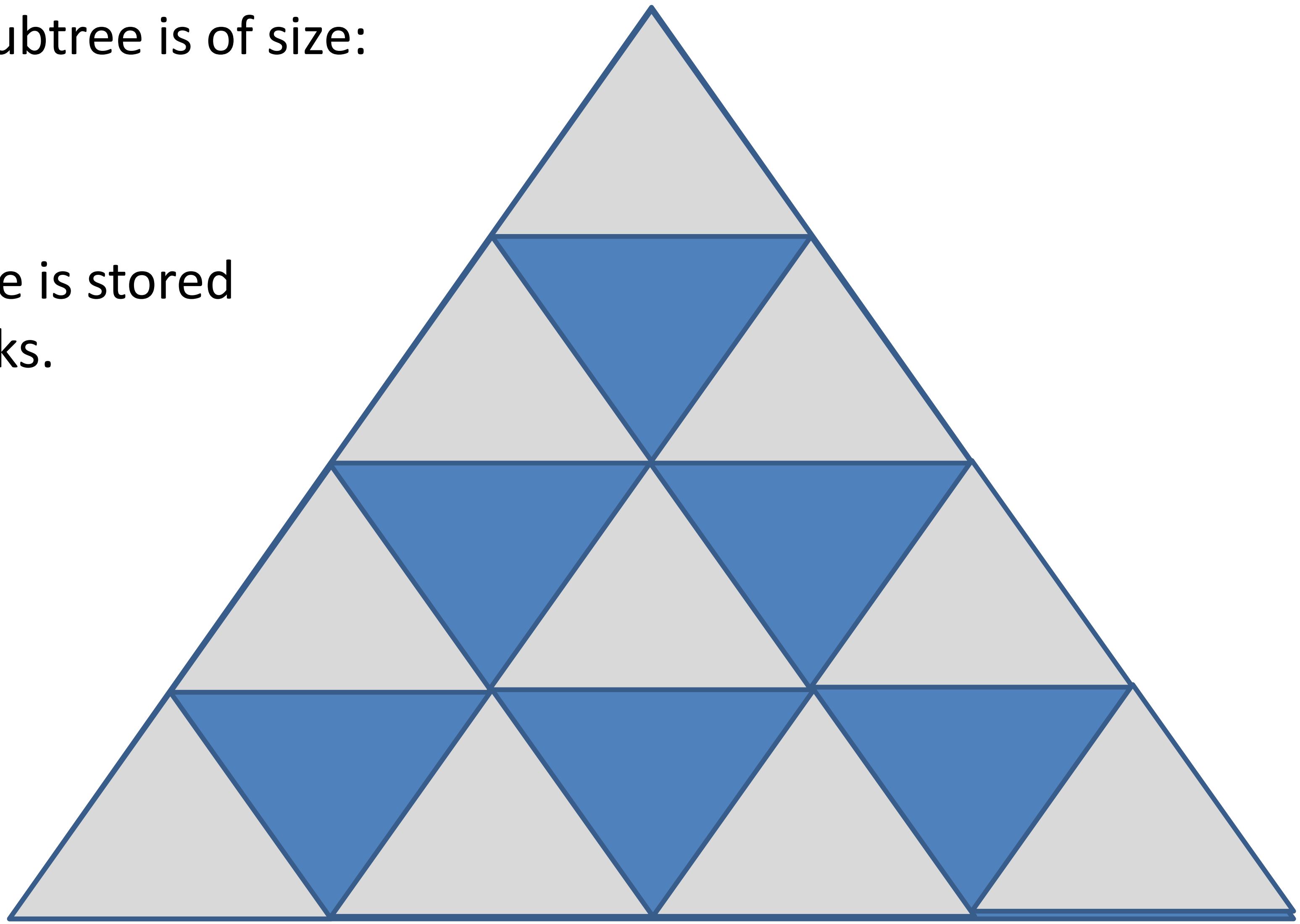
van Emde Boas Layout: Analysis

Run the recursive decomposition until each subtree is of size:

> $\sqrt{B}$

< B

Each subtree is stored in O(1) blocks.



van Emde Boas Layout: Analysis

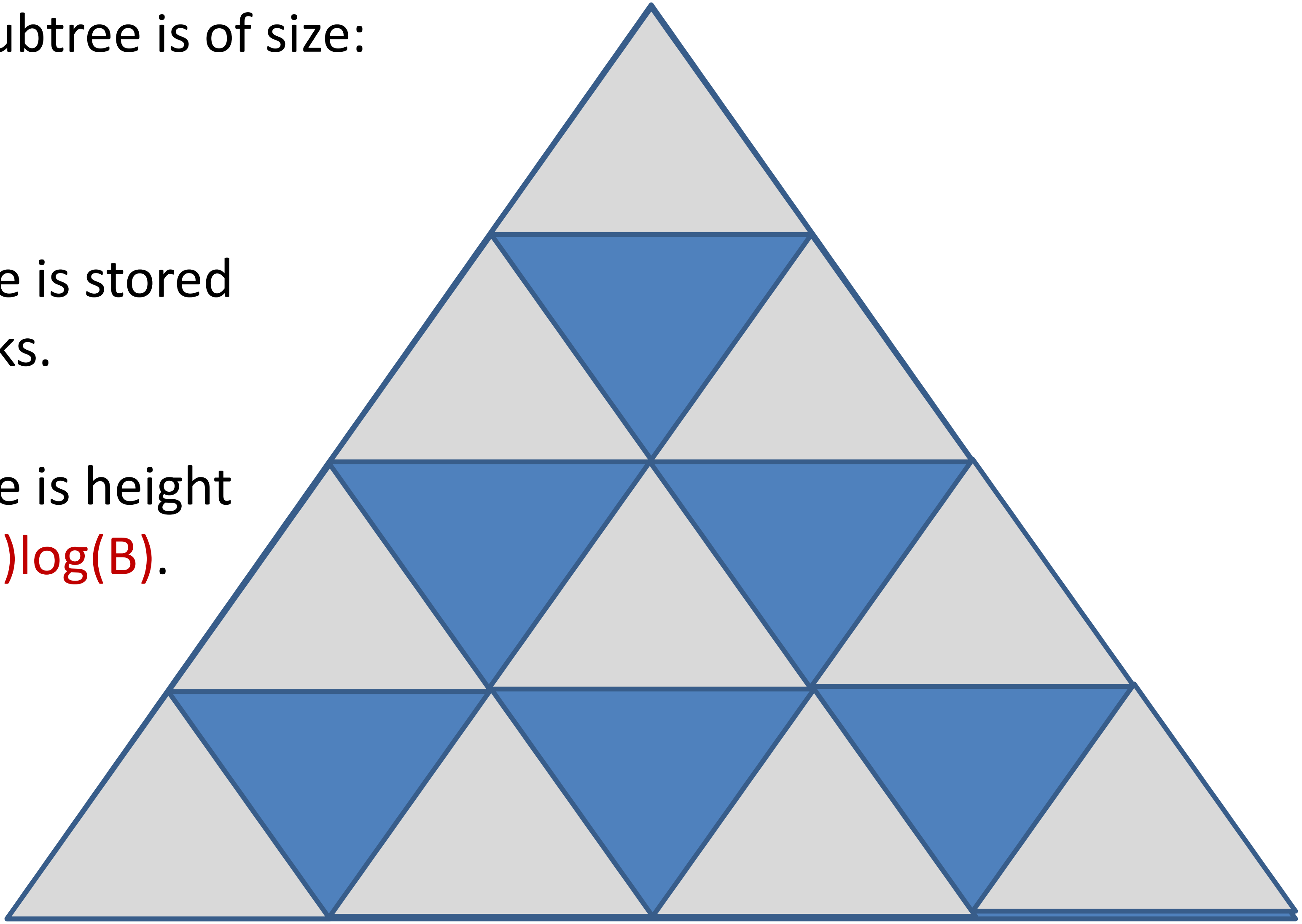
Run the recursive decomposition until each subtree is of size:

> $\sqrt{B}$

< B

Each subtree is stored in O(1) blocks.

Each subtree is height at least (1/2)log(B).



van Emde Boas Layout: Analysis

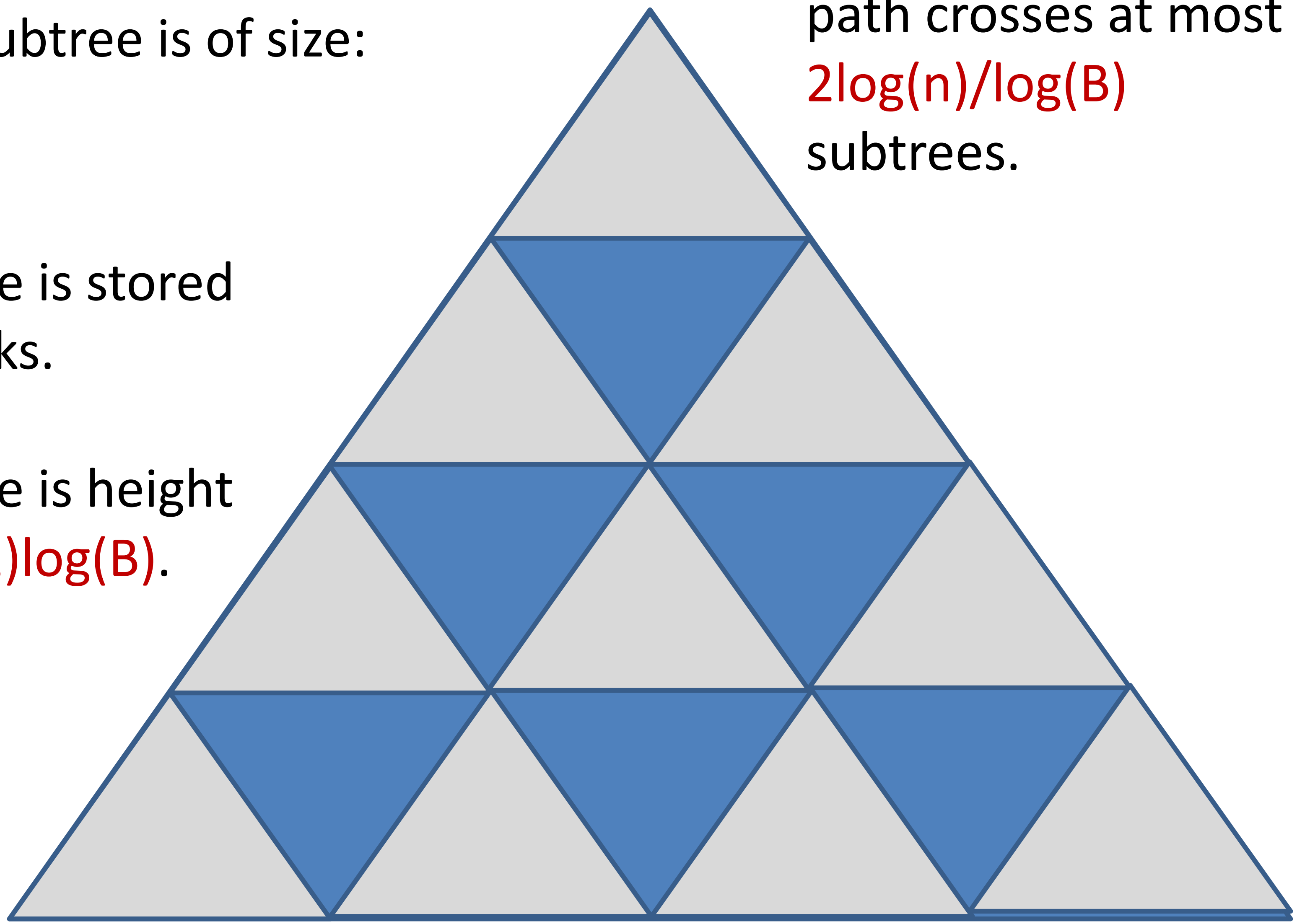
Run the recursive decomposition until each subtree is of size:

> $\sqrt{B}$

< B

Each subtree is stored in O(1) blocks.

Each subtree is height at least (1/2)log(B).



Any root-to-leaf path crosses at most 2log(n)/log(B) subtrees.

van Emde Boas Layout: Analysis

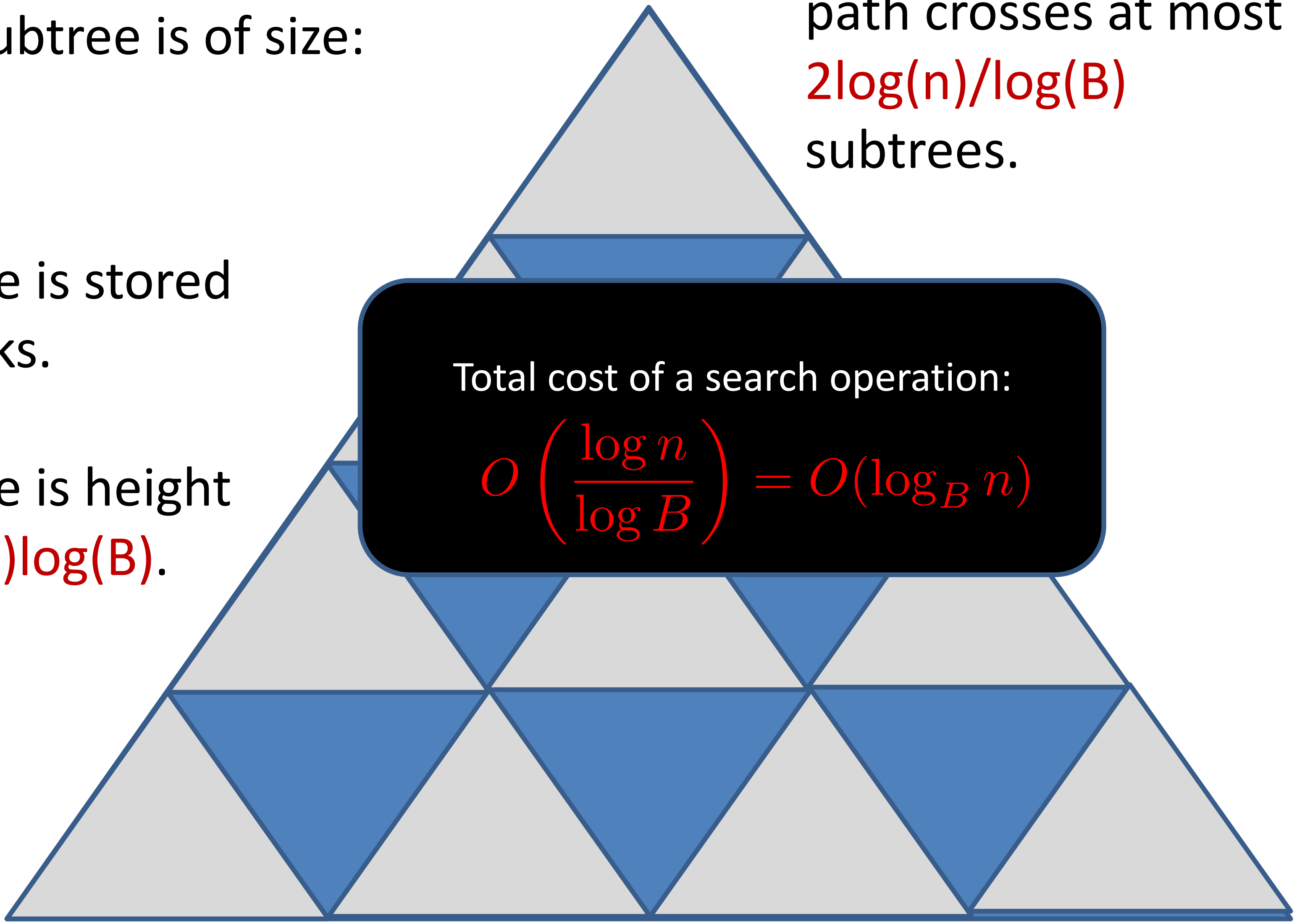
Run the recursive decomposition until each subtree is of size:

- > √B
- < B

Each subtree is stored in O(1) blocks.

Each subtree is height at least (1/2)log(B).

Any root-to-leaf path crosses at most 2log(n)/log(B) subtrees.



Total cost of a search operation:

$$O\left(\frac{\log n}{\log B}\right) = O(\log_B n)$$

Today's Plan

Searching and Sorting

1. B-trees
   - ⇒ Algorithm
   - ⇒ Amortized analysis

2. Buffer trees
   - ⇒ Write-optimized data structures
   - ⇒ Buffered data structures
   - ⇒ Amortized analysis

3. van Emde Boas Search Tree
   - ⇒ Cache-oblivious algorithms
   - ⇒ van Emde Boas memory layout

Questions

Buffer tree:

What if degree of each node is increased to: $\sqrt{B}$

What if degree of each node is increased to: B^{ϵ}

Sorting:

Design a Buffer Tree that is good for sorting. (Hint: you can make the degree bigger, the buffer bigger, and/or the leaves bigger.)

Goal: $O\left(\frac{n}{B}\log_{M/B}\frac{n}{B}\right)$

More sorting:

Design an external memory MergeSort algorithm.

(Hint: you need to merge more efficiently.)

(Hint 2: you will need to do a multiway merge.)

Goal: $O\left(\frac{n}{B}\log_{M/B}\frac{n}{B}\right)$