CS 240 – Data Structures and Data Management

Module 11E: External Memory - enriched

T. Biedl E. Kondratovsky M. Petrick O. Veksler
Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2022

version *2022−03−29 14:38*

Outline

11 External Memory
- Red-black trees
- Pre-emptive splitting/merging
- B^+-trees
- LSM-trees
- Extendible Hashing

Outline

11 External Memory
- Red-black trees
- Pre-emptive splitting/merging
- B^+-trees
- LSM-trees
- Extendible Hashing

Towards red-black-tree

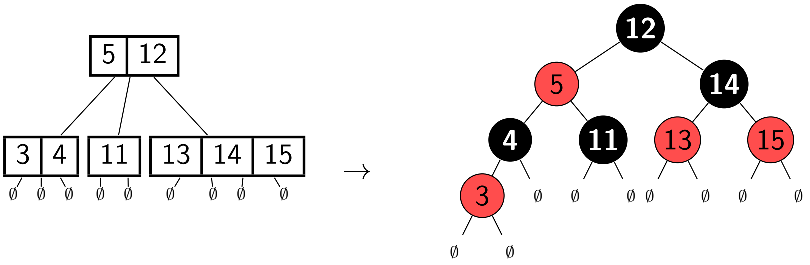
(We currently only consider run-time in RAM. We will return to the EMM shortly.)

- Recall: All operations in 2-4 trees have $O(\log n)$ worst-case run-time.
- The height is much smaller than for AVL-trees ($\log_2(\frac{n+1}{2})$ vs. $\log_\Phi(n) \approx 1.44 \log_2 n$.)
- So they might be more efficient, depending on implementation details.

- But: Handling three kinds of nodes is cumbersome.
  (We either need a list for KVPs and subtrees, or waste space at nodes to have space for links always available.)

Better idea: Design a class of binary search trees that mirrors 2-4-trees!

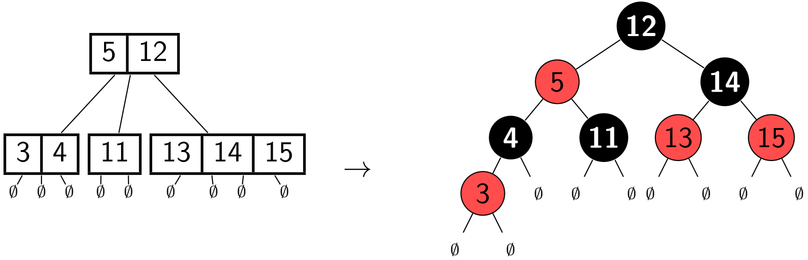
2-4-tree to red-black-tree



Converting a 2-4-tree:

- A d-node becomes a black node with $d-1$ red children
  (Assembled so that they form a BST of height at most 1.)

2-4-tree to red-black-tree



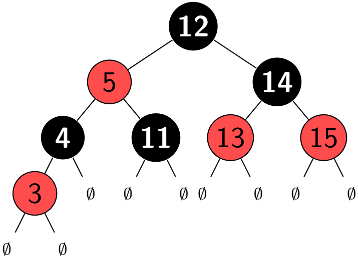
Converting a 2-4-tree:

- A d-node becomes a black node with $d-1$ red children (Assembled so that they form a BST of height at most 1.)

Resulting properties:

- Any red node has a black parent.
- Any empty subtree T has the same **black-depth** (number of black nodes on path from root to T)

Red-black-trees



Definition: A **red-black tree** is a binary search tree such that

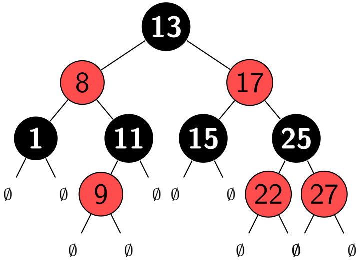
- Every node has a color (red or black)
- Every red node has a black parent.
  (In particular the root is black.)
- Any empty subtree T has the same black-depth.

Note: Can store this with one bit overhead per node.

Red-black tree

Rather than proving properties directly, we re-use properties of 2-4-trees.

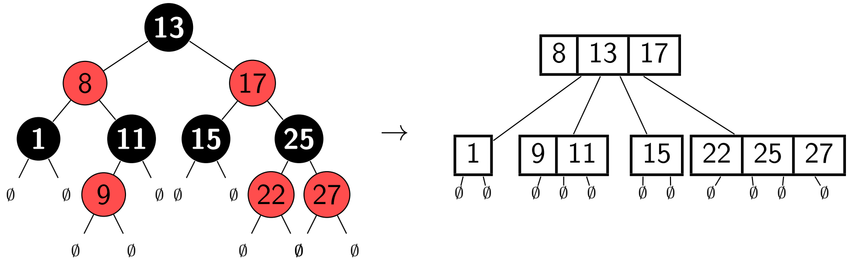
Lemma: Any red-black tree T can be converted into a 2-4-tree T' where $\mathit{height}(T') = \mathit{black\text{-}depth}(T) - 1$.



Red-black tree

Rather than proving properties directly, we re-use properties of 2-4-trees.

Lemma: Any red-black tree T can be converted into a 2-4-tree T' where $height(T') = black\text{-}depth(T) - 1$.



Proof:

- Black node with $0 \le d \le 2$ red children becomes a $(d{+}1)$-node

Red-black tree properties

- Red-black trees have height $\leq 2\log(\frac{n+1}{2}) + 1$
  - black-depth $\leq \log(\frac{n+1}{2}) + 1$ by 2-4-tree height.
  - At least half of the nodes on the path to deepest nodes are black (recall: red nodes have black parents)

$\Rightarrow$ height$=$# nodes on path - 1 $\leq$ 2 black-depth - 1

Red-black tree properties

- Red-black trees have height $\leq 2\log(\frac{n+1}{2}) + 1$
  - black-depth $\leq \log(\frac{n+1}{2}) + 1$ by 2-4-tree height.
  - At least half of the nodes on the path to deepest nodes are black (recall: red nodes have black parents)

  $\Rightarrow$ height$=\#$ nodes on path - 1 $\leq$ 2 black-depth - 1

- *insert*/*delete* can be done as for 2-4-trees.
  - One can "translate" the code directly to red-black trees.
  - The transfer/split/merge operations become rotations.
- So all operations take $\Theta(\log n)$ worst-case time.
- In the worst case, $\Theta(\log n)$ rotations are required for *insert*/*delete*.
- But experiments show that few rotations usually suffice, and updates are faster in red-black trees than in AVL-trees.

Red-black tree properties

- Red-black trees have height $\leq 2\log(\frac{n+1}{2}) + 1$
  - black-depth $\leq \log(\frac{n+1}{2}) + 1$ by 2-4-tree height.
  - At least half of the nodes on the path to deepest nodes are black (recall: red nodes have black parents)
  - $\Rightarrow$ height=# nodes on path - 1 $\leq$ 2 black-depth - 1

- *insert*/*delete* can be done as for 2-4-trees.
  - One can "translate" the code directly to red-black trees.
  - The transfer/split/merge operations become rotations.
- So all operations take $\Theta(\log n)$ worst-case time.
- In the worst case, $\Theta(\log n)$ rotations are required for *insert*/*delete*.
- But experiments show that few rotations usually suffice, and updates are faster in red-black trees than in AVL-trees.

This is a very efficient balanced binary search tree.

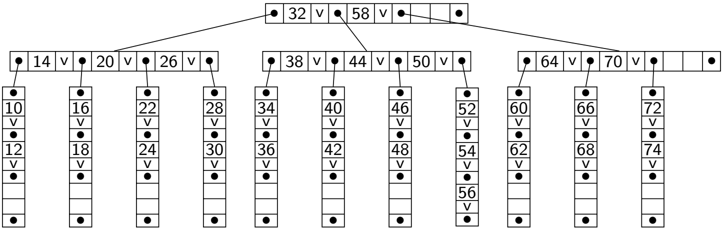
(There are even better balanced binary search trees. No details.)

Outline

11 External Memory

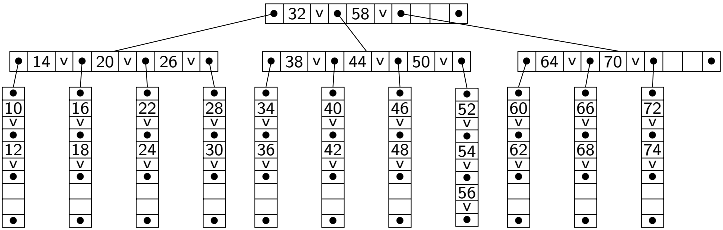
- Red-black trees
- Pre-emptive splitting/merging
- B^+-trees
- LSM-trees
- Extendible Hashing

Pre-emptive splitting/merging



- Observe: *BTree::insert*(k, v) traverses tree twice:
  - Search down on a path to the leaf where we add (k, v).
  - Go back up on the path to fix overflow, if needed.
- So the number of block-transfers could be twice the height.
- How can we avoid this?

Pre-emptive splitting/merging

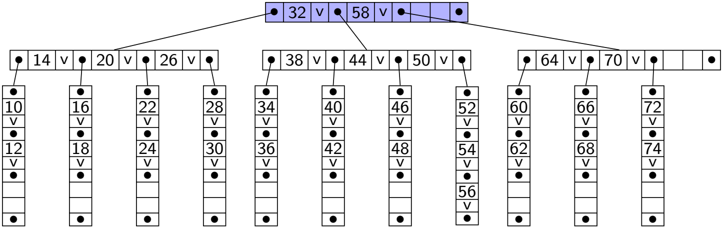


- Observe: *BTree::insert*(k, v) traverses tree twice:
  - Search down on a path to the leaf where we add (k, v).
  - Go back up on the path to fix overflow, if needed.
- So the number of block-transfers could be twice the height.
- How can we avoid this?

- **Idea:** During the search, *always* split if the node is full.
- Then a node split at the leaf does not create an overfull parent.

Pre-emptive splitting/merging example

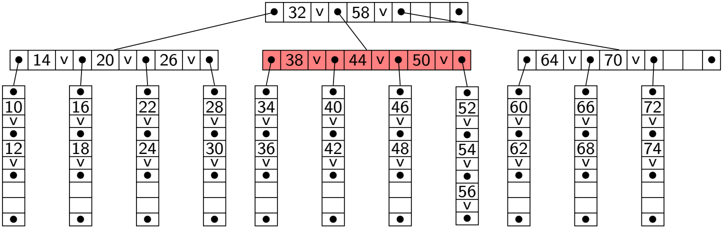
PreemptiveBTree::insert(49):



- If node is not full, keep searching.

Pre-emptive splitting/merging example

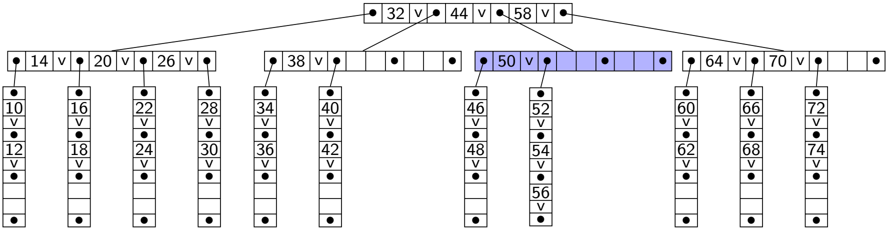
PreemptiveBTree::insert(49):



- If node is not full, keep searching.
- If node is full, immediately split.

Pre-emptive splitting/merging example

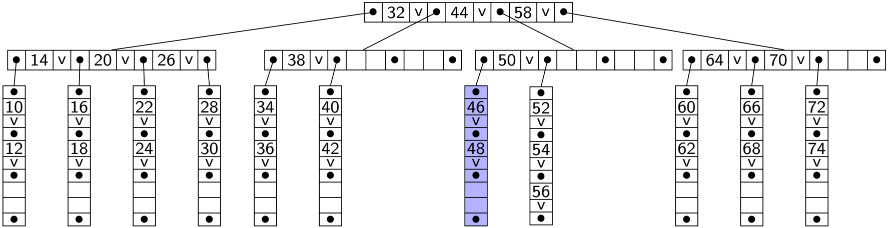
PreemptiveBTree::insert(49):



- If node is not full, keep searching.
- If node is full, immediately split.
- Then keep searching in appropriate new node.

Pre-emptive splitting/merging example

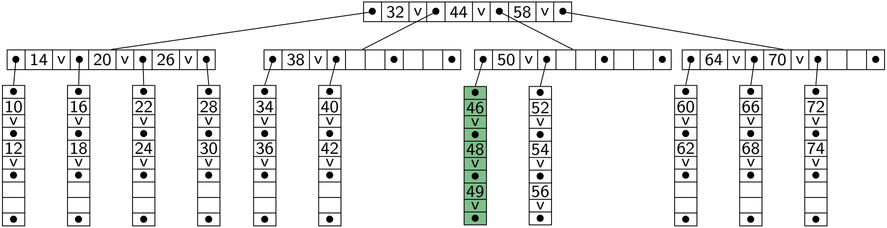
PreemptiveBTree::insert(49):



- If node is not full, keep searching.
- If node is full, immediately split.
- Then keep searching in appropriate new node.
- We may have split unnecessarily. (But space is cheap.)

Pre-emptive splitting/merging example

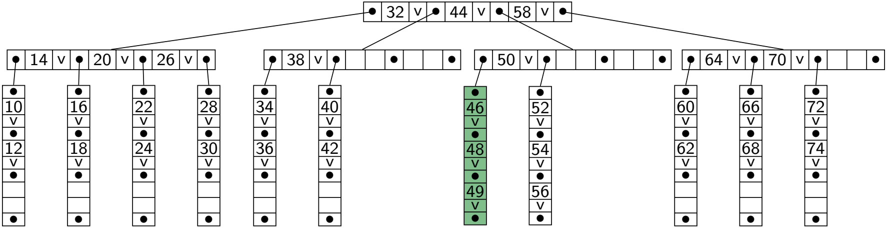
PreemptiveBTree::insert(49):



- If node is not full, keep searching.
- If node is full, immediately split.
- Then keep searching in appropriate new node.
- We may have split unnecessarily. (But space is cheap.)

Pre-emptive splitting/merging example

PreemptiveBTree::insert(49):



- If node is not full, keep searching.
- If node is full, immediately split.
- Then keep searching in appropriate new node.
- We may have split unnecessarily. (But space is cheap.)

- Similarly *delete* should pre-emptively merge. (No details.)
- With this, we no longer need parent-references.

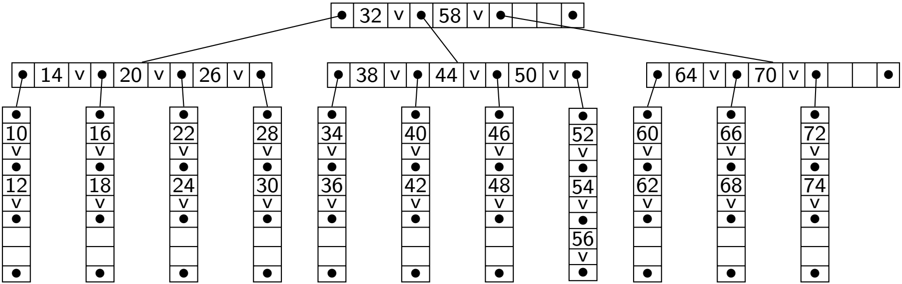
Outline

11 External Memory

- Red-black trees
- Pre-emptive splitting/merging
- B^+-trees
- LSM-trees
- Extendible Hashing

Towards B^+-trees

In a B-tree, each node is one block of memory. In this example, up to 10 keys/references fit into one block, so the order is 4.



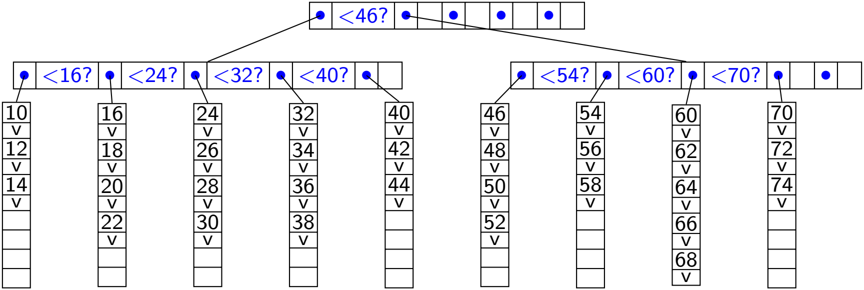
This B-tree could store up to 63 KVPs with height 2.

Two ideas to achieve smaller height:

1. The leaves are wasting space for references that will never be used.
2. Use a *decision-tree version* $\Rightarrow$ inner nodes can have more children.

B^+-trees

- Each node is one block of memory.
- All KVPs are stored at *leaves*. Each leaf is at least half full.
- *Interior nodes* store only keys for comparison during search.
- Interior (non-root) nodes have at least half of the possible subtrees.
- *insert*/*delete* use pre-emptive splitting/merging.

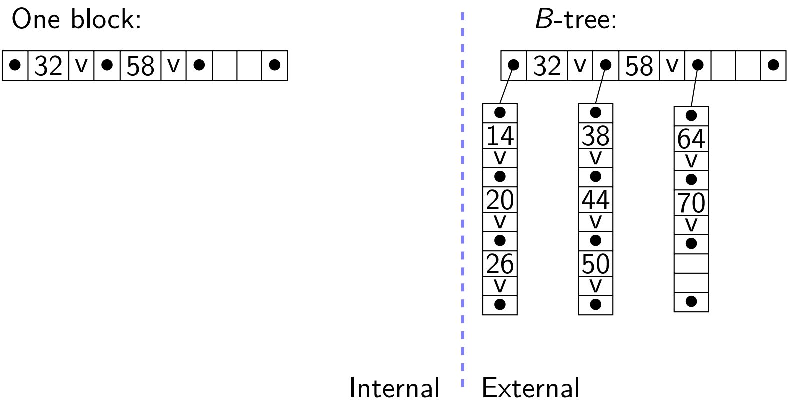


This B^+-tree could store up to 125 KVPs with height 2.

Outline

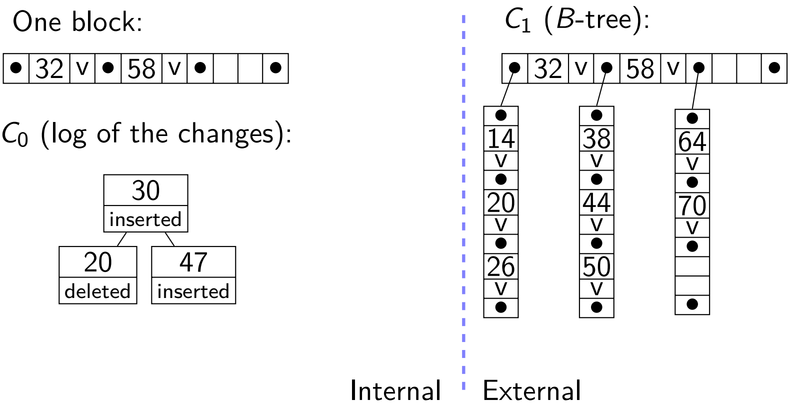
11 External Memory
- Red-black trees
- Pre-emptive splitting/merging
- B^+-trees
- LSM-trees
- Extendible Hashing

Towards LSM-trees



- Main memory only requires 1-2 blocks at a time.
- Roughly $M - 2B$ space free.
- How can we use this to increase speed for updates?

LSM-trees



- Store dictionary in internal memory that logs all changes
- To search: first search in C_0, then (if needed) in C_1
- If internal memory full: do lots of updates in C_1 at once

Outline

11 External Memory

- Red-black trees
- Pre-emptive splitting/merging
- B^+-trees
- LSM-trees
- Extendible Hashing

Dictionaries for Integers in External Memory

Recall Hashing:

- Direct Addressing allowed for $O(1)$ insert and delete if keys are small integers.
- If keys are too big, use hash-function to map them to (smaller) integers.
- Expected run-time of operations is $O(1)$ if load factor α is kept small

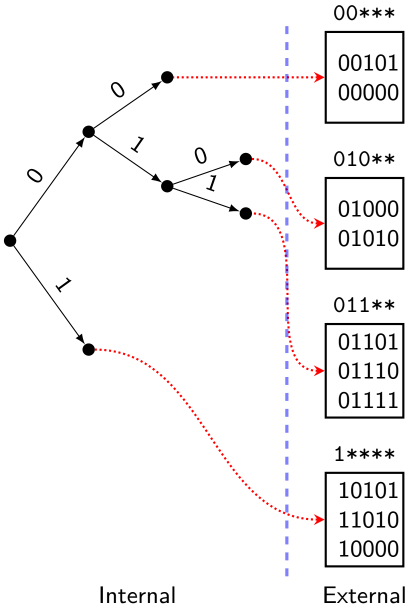
This does not adapt well to external memory.

- We must occasionally re-hash to keep α small.
- And re-hashing must load *all* n/B blocks.
- This is unacceptably slow.

Goal: Data structure for integers that typically uses $O(1)$ block transfers, and never needs to load all blocks.

Idea: Hash-values = bitstrings. Store trie of links to blocks of integers.

Trie of blocks – Overview



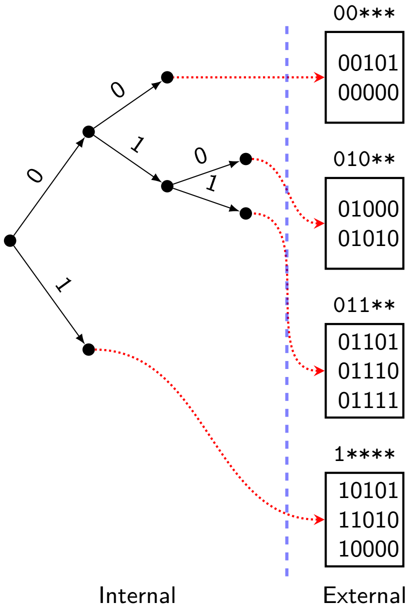
Assumption: We store non-negative integers (here written as bitstrings). [Typically these are hash-values.]

Build trie D (the **directory**) of integers in internal memory.

Stop splitting in trie when remaining items fit in one block. (∼ pruned trie, but stop earlier)

Each leaf of D refers to block of external memory that stores the items.

Trie of blocks – operations



search(k): Search for k in D until we reach leaf ℓ. Load block at ℓ and search in it.
1 block transfer.

insert(k): Search for k, load block, then insert k. If this exceeds block-capacity, split at trie-node and split blocks (possibly repeatedly).
Typically 2 **block transfers**.

delete(k): Search for k, load block, then delete k.
Optional: combine underfull blocks.
2 block transfers.

Trie of blocks: Insert

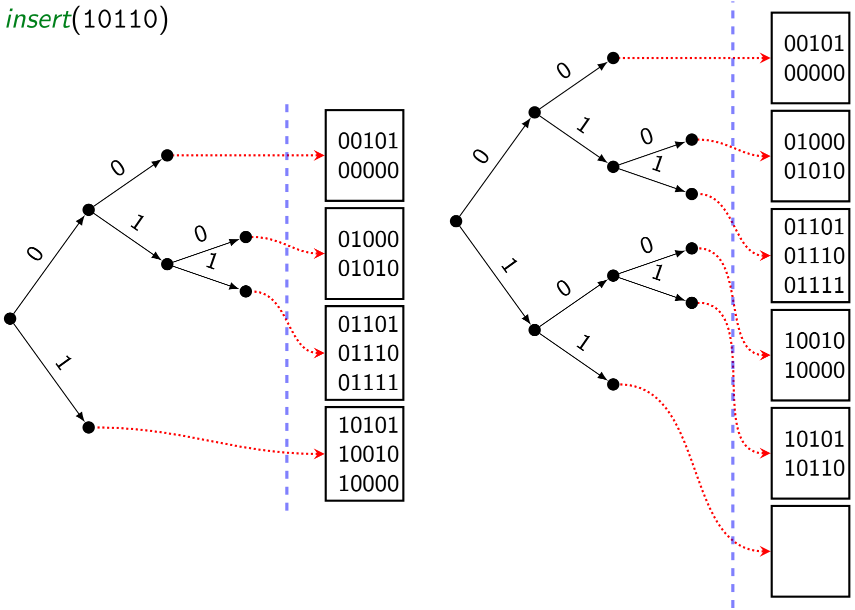
TrieOfBlocks::insert(k, v)
(k, v): key-value pair, k is a bit-string

$\ell \leftarrow$ *Trie::search*(D, k) // leaf with prefix of k
$d \leftarrow$ depth of ℓ in D
transfer block P that ℓ refers to
while P has no room for additional items
  Split P into two blocks P_0 and P_1 by $k[d]$
  Create two children ℓ_0 and ℓ_1 of ℓ, linked to P_0 and P_1
  $d \leftarrow d{+}1, \ell \leftarrow \ell_{k[d]}, P \leftarrow P_{k[d]}$
insert (k, v) into P

Note: This may create empty blocks, but this should be rare.

Trie of blocks: Insert

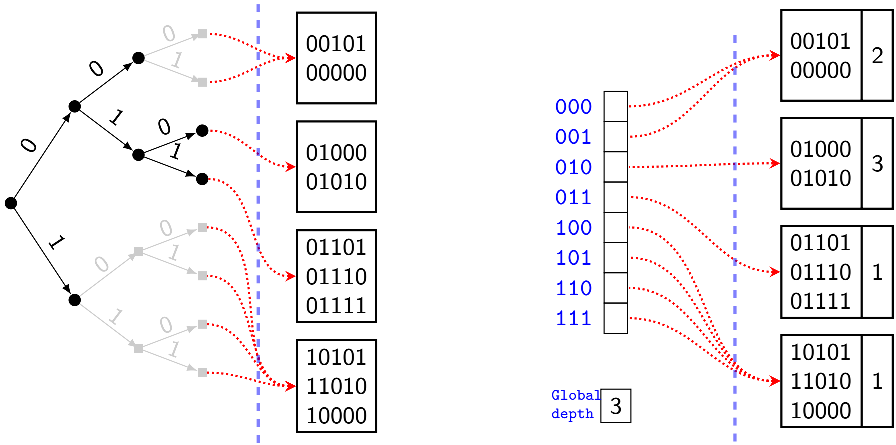
insert(10110)



Extendible hashing

We can save links (hence space in internal memory) with two tricks:

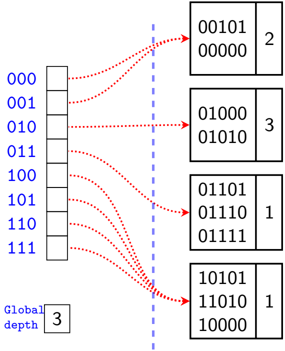
- Expand the trie so that all leaves have the same **global depth** d_D.
- Store *only* the leaves, and in an array D of size 2^{d_D}.

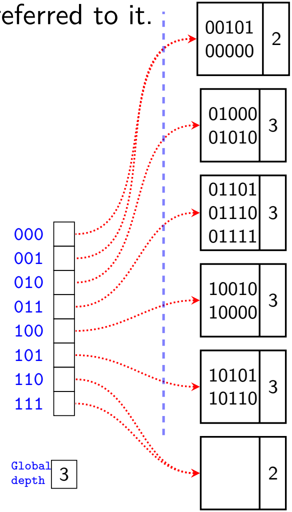


Extendible hashing operations

- Conceptually: convert table to trie, do operation, convert trie to table
- But work directly on table if each block stores its **local depth**, i.e., the depth of the original trie-node that referred to it.

Example: *insert*(10110)

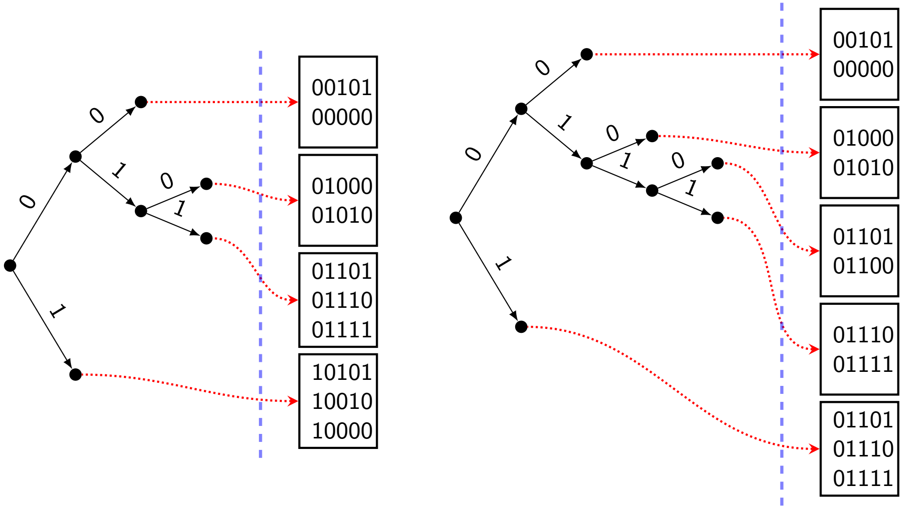




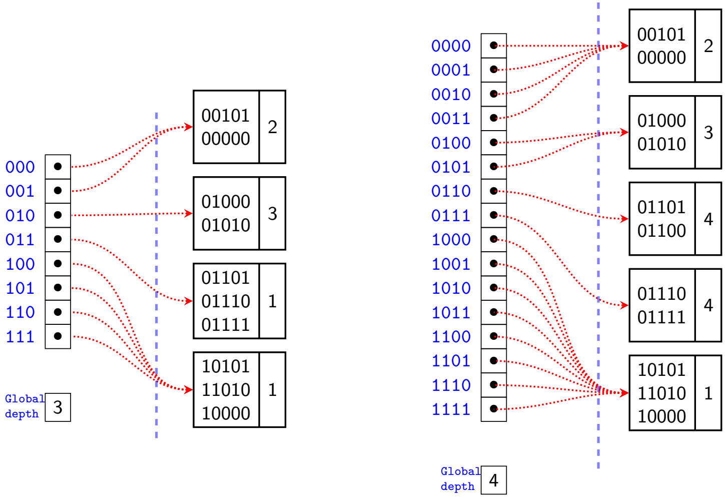
Extendible hashing operations

If *insert* increased the trie-height, then the array-size now doubles.

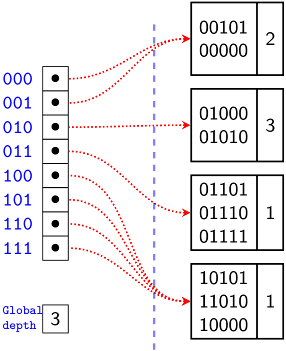
Example: *insert*(01100) in trie of blocks

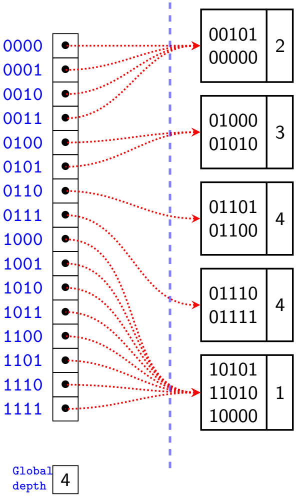


insert(01100) in extendible hash-table



insert(01100) in extendible hash-table





But notice: We do *not* need to load extra blocks for this.
The number of block-transfers is exactly the same as with the trie of blocks, but the space used by the dictionary is much better.

Extendible hashing discussion

- Hashing collisions (= duplicate keys) are resolved within the block and do not affect the block transfers.
  If more items collide than can fit into a block we *extend the hash-function*, i.e., make bit-strings longer without changing the initial bits.
- Directory typically fits into in internal memory.
  If it does not, then strategies similar to B-trees can be applied.
- Only 1 or 2 block transfers expected for *any* operation.
- To make more space, we only add one block.
  Rarely change the size of the directory.
  Never have to move all items. (in contrast to re-hashing!)
- Space usage is not too inefficient: one can show that under uniform distribution assumption each block is expected to be 69% full.