CS 240 – Data Structures and Data Management

Module 11: External Memory - enriched

T. Biedl   É. Schost   O. Veksler
Based on lecture notes by many previous cs240 instructors

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

Winter 2021

version *2021–03–30 19:23*

Outline

1. External Memory
   - Red-black trees
   - Pre-emptive splitting/merging
   - B^+-trees

Outline

1. External Memory
   - Red-black trees
   - Pre-emptive splitting/merging
   - B^+-trees

Towards red-black-tree

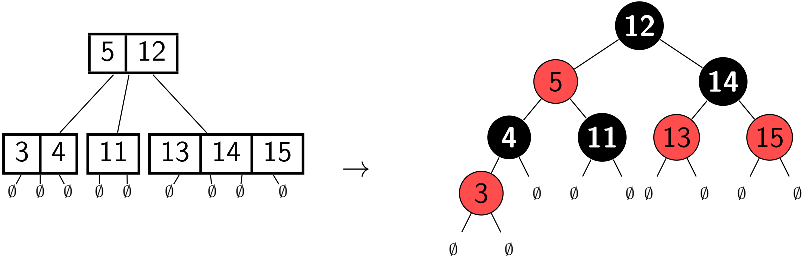
(We currently only consider run-time in RAM. We will return to the EMM shortly.)

- 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!

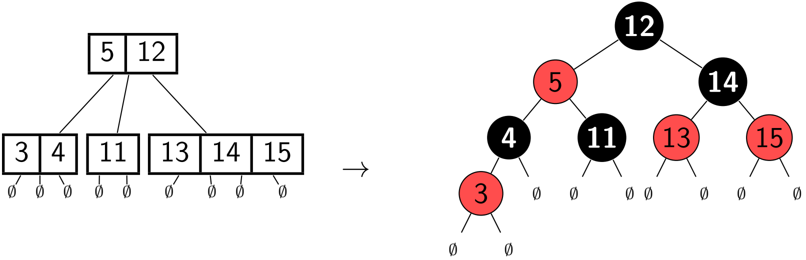
2-4-tree to red-black-tree



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



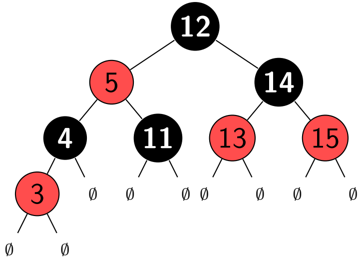
Converting a 2-4-tree:

- 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

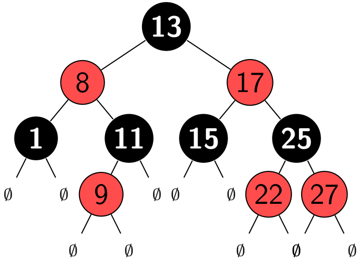
- Every node has a color (red or black)
- Every red node has a black parent.
  (In particular the root is black.)
- Any empty subtree T has the same black-depth.

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.

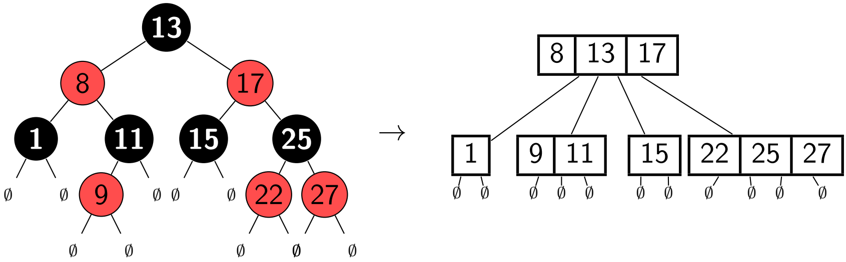
Lemma: Any red-black tree T can be converted into a 2-4-tree T' where $height(T') = black\text{-}depth(T) - 1$.



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 $0 \leq d \leq 2$ 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 red-black trees are faster than 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 red-black trees are faster than AVL-trees.

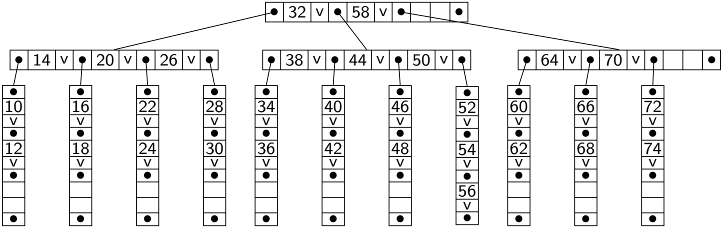
This is a very efficient balanced binary search tree.

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

Outline

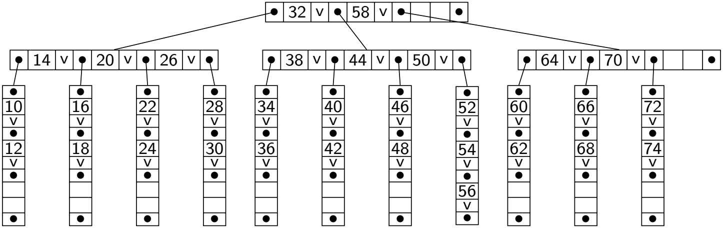
1. External Memory
   - Red-black trees
   - Pre-emptive splitting/merging
   - B^+-trees

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

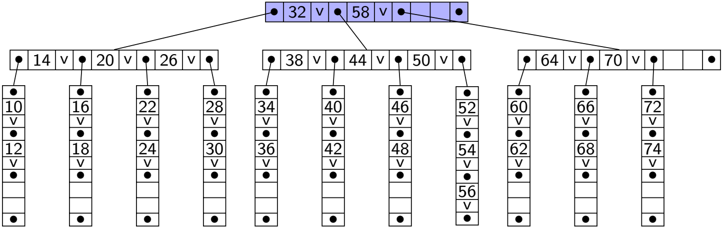


- Observe: *BTree::insert*(k, v) traverses tree twice:
  - Search down on a path to the leaf where we add (k, v).
  - Go back up on the path to fix overflow, if needed.
- So the number of block-transfers could be twice the height.
- How can we avoid this?

- **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

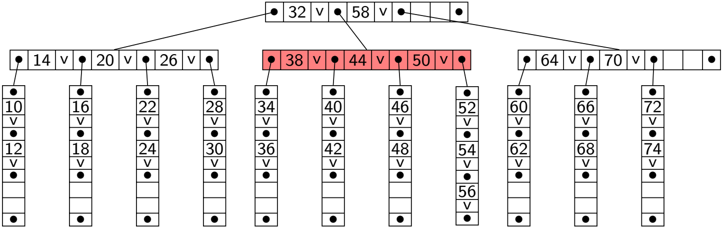
PreemptiveBTree::insert(49):



- If node is not full, keep searching.

Pre-emptive splitting/merging example

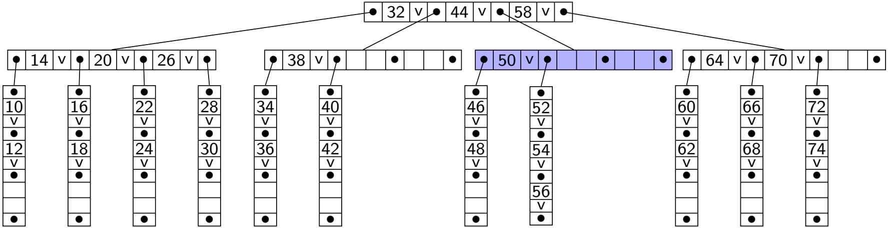
PreemptiveBTree::insert(49):



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

Pre-emptive splitting/merging example

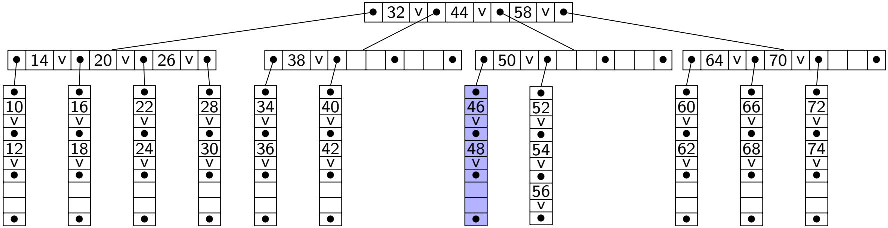
PreemptiveBTree::insert(49):



- 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

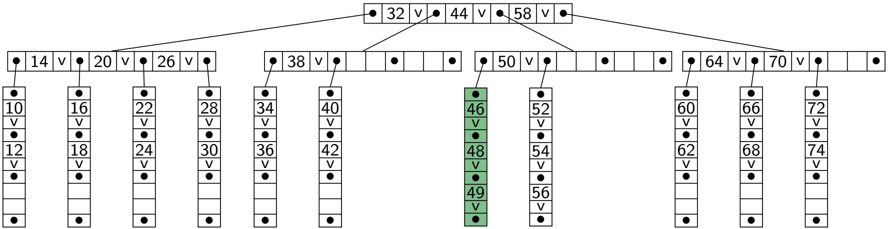
PreemptiveBTree::insert(49):



- 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

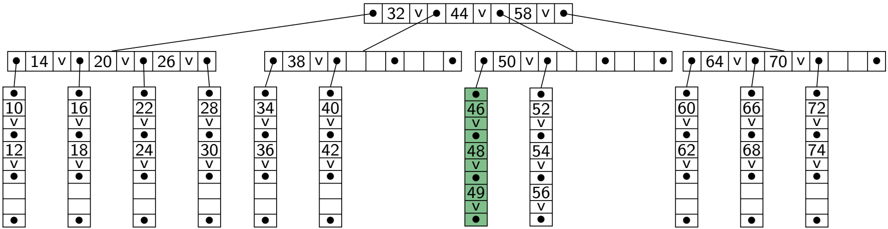
PreemptiveBTree::insert(49):



- 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

1. External Memory
   - Red-black trees
   - Pre-emptive splitting/merging
   - B^+-trees

Towards B^+-trees

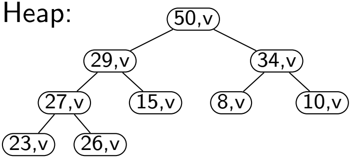
- Two types of tree-structures, depending on where values are stored.

Towards B^+-trees

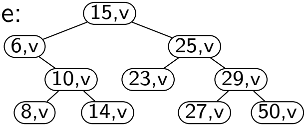
- Two types of tree-structures, depending on where values are stored.

Storage-variant: Every node stores a KVP.

Heap:



BST-tree:

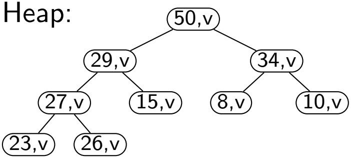


Towards B^+-trees

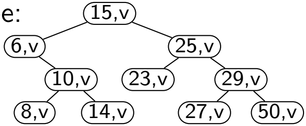
- Two types of tree-structures, depending on where values are stored.

Storage-variant: Every node stores a KVP.

Heap:

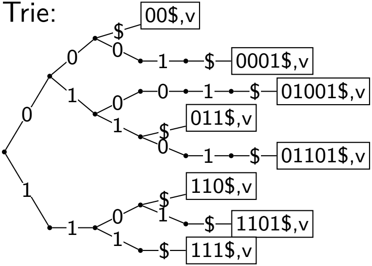


BST-tree:

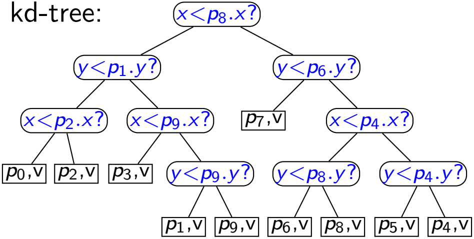


Decision-variant: All KVPs at leaves, internal nodes/edges guide search.

Trie:

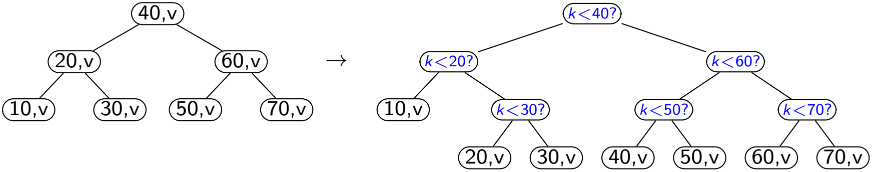


kd-tree:



Towards B^+-trees

- For storage-variant, there usually exists an equivalent decision-variant.

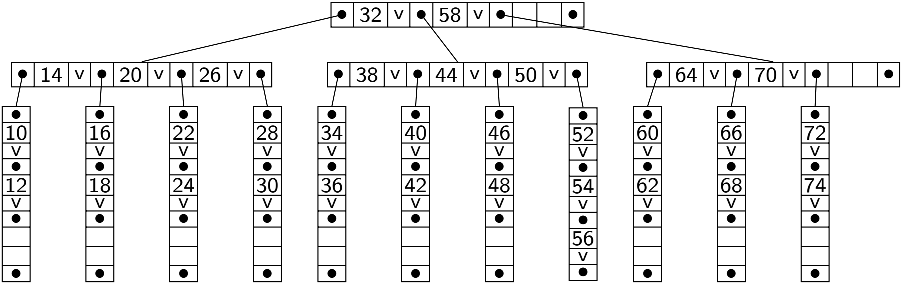


- For example for binary search trees:
  - Choose a tree with n leaves where internal nodes have 2 children.
  - Internal nodes store minimum in right subtree.
  - Rotations now also update split-lines.

  We have seen a similar construction in priority search trees.
- In *internal memory*, decision-tree variants waste space (typically $\approx$ twice as many nodes)

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.



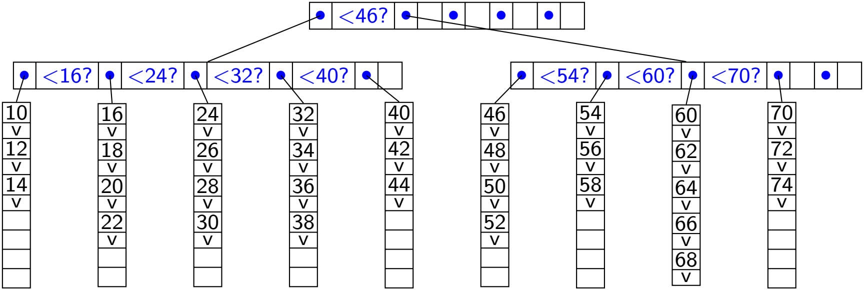
This B-tree could store up to 63 KVPs with height 2.

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

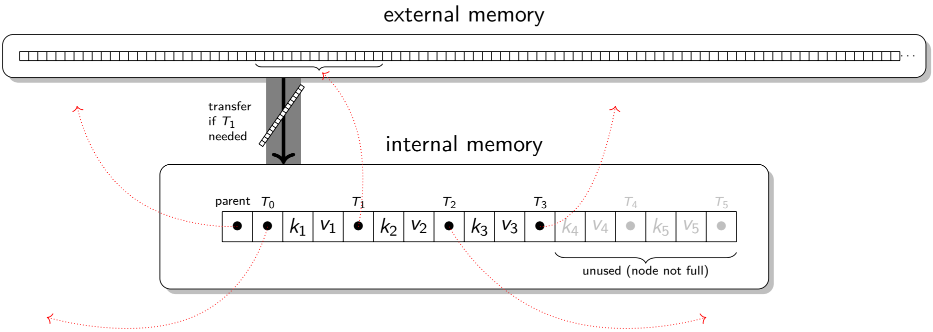
- Each node is one block of memory.
- All KVPs are stored at *leaves*. Each leaf is at least half full.
- *Interior nodes* store only keys for comparison during search.
- Interior (non-root) nodes have at least half of the possible subtrees.
- *insert*/*delete* use pre-emptive splitting/merging.



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

B^+-trees in external memory

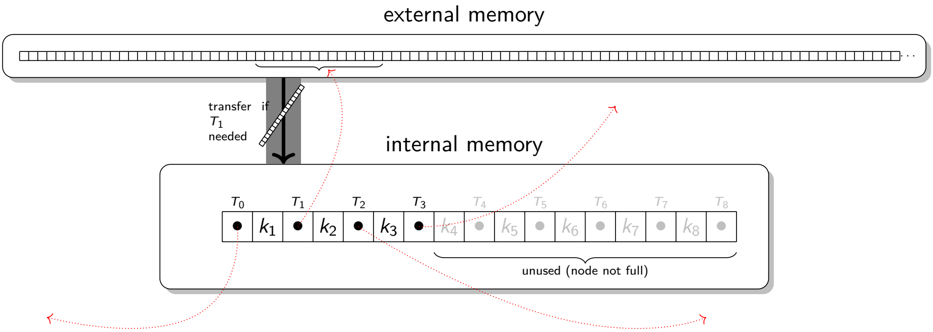
Recall: Close-up on one node of a regular B-tree:



In this example: 17 computer-words fit into one block, so the B-tree can have order 6.

B^+-tree in external memory

Contrast with: Close-up on one interior node of a B^+-tree:



In this example: 17 computer-words fit into one block, so the B^+-tree can have order 9.

B^+-tree summary

- Order is typically a factor of $\frac{3}{2}$ bigger than for B-trees.
- B^+-tree needs to store $\approx$ twice as many keys

B^+-tree summary

- Order is typically a factor of $\frac{3}{2}$ bigger than for B-trees.
- B^+-tree needs to store $\approx$ twice as many keys
- Height-comparison (where b is the order of the B-tree):

B^+-tree	vs.	B-tree
$\log_{\frac{3}{2}b}(2n)$		$\log_b(n)$

B^+-tree summary

- Order is typically a factor of $\frac{3}{2}$ bigger than for B-trees.
- B^+-tree needs to store $\approx$ twice as many keys
- Height-comparison (where b is the order of the B-tree):

B^+-tree	vs.	B-tree
$\log_{\frac{3}{2}b}(2n)$		$\log_b(n)$
$\Vert$		$\Vert$
$\dfrac{\log n + 1}{\log b + \underbrace{\log(3/2)}_{\approx 0.7}}$	$<$	$\dfrac{\log n}{\log b}$

B^+-tree summary

- Order is typically a factor of $\frac{3}{2}$ bigger than for B-trees.
- B^+-tree needs to store $\approx$ twice as many keys
- Height-comparison (where b is the order of the B-tree):

B^+-tree	vs.	B-tree
$\log_{\frac{3}{2}b}(2n)$		$\log_b(n)$
$\Vert$		$\Vert$
$\dfrac{\log n + 1}{\log b + \underbrace{\log(3/2)}_{\approx 0.7}}$	$<$	$\dfrac{\log n}{\log b}$

- B^+-trees have smaller height, and use only one pass.
- Best for storing huge dictionaries in external memory.

(For data base implementations, there are further tricks such as linking the leaves as a list. See cs448 for details.)