Module 11: External memory

CS 240 - Data Structures and Data Management

Mark Petrick

Based on lecture notes by many previous cs240 instructors

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

Fall 2017

Different levels of memory

Current architectures:

- registers (very fast, very small)
- cache L1, L2 (still fast, less small)
- main memory
- external memory: disk (slow, very large)

General question: how to adapt our algorithms to take the memory hierarchy into account, avoiding transfers as much as possible?

Review: dictionary ADT

A *dictionary* is a collection of *items*, each of which contains

- a *key*
- some *data*,

and is called a *key-value pair* (KVP). Keys can be compared and are (typically) unique.

Operations:

- *search*(k)
- *insert*(k, v)
- *delete*(k)
- optional: *join*, *isEmpty*, *size*, *etc.*

Dictionaries in external memory

Tree-based data structures have poor *memory locality*:
If an operation accesses m nodes, then it must access
m spaced-out memory locations.

Observation: Accessing a single location in *external memory* (e.g. hard disk) automatically loads a whole block (or "page").

- In an AVL tree, $\Theta(\log n)$ pages are loaded in the worst case.
- Better solution: B-trees

2-3 Trees

A 2-3 Tree is like a BST with additional structual properties:

- Every internal node either contains *one KVP* and *two children*, or *two KVPs* and *three children*.
- The leaves are *NIL* (do not store keys)
- All the leaves are at the same level.

Searching through a 1-node is just like in a BST.
For a 2-node, we must examine both keys and follow the appropriate path.

Insertion in a 2-3 tree

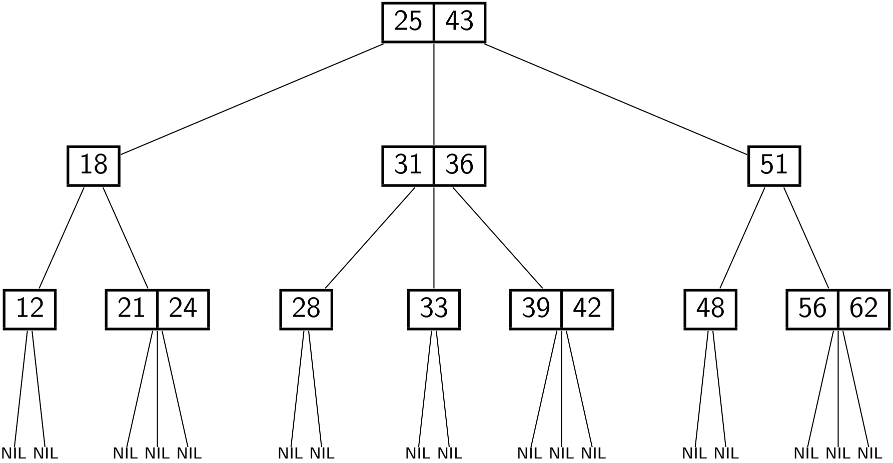
First, we search to find the lowest internal node where the new key belongs.

If the node has only 1 KVP, just add the new one to make a 2-node.

Otherwise, order the three keys as $a < b < c$.
Split the node into two 1-nodes, containing a and c,
and (recursively) insert b into the parent along with the new link.

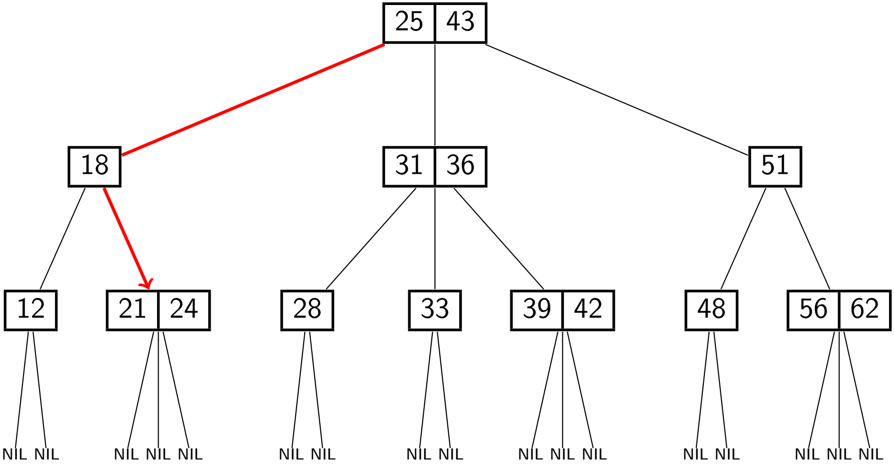
2-3 Tree Insertion

Example: *insert*(19)



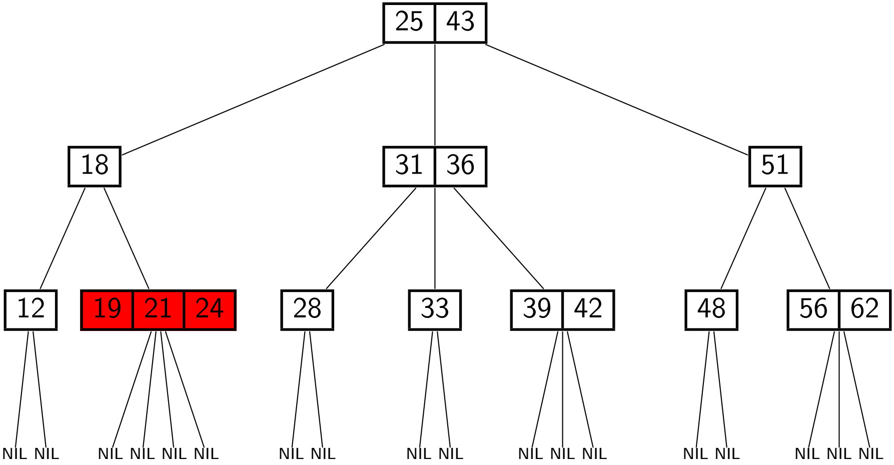
2-3 Tree Insertion

Example: *insert*(19)



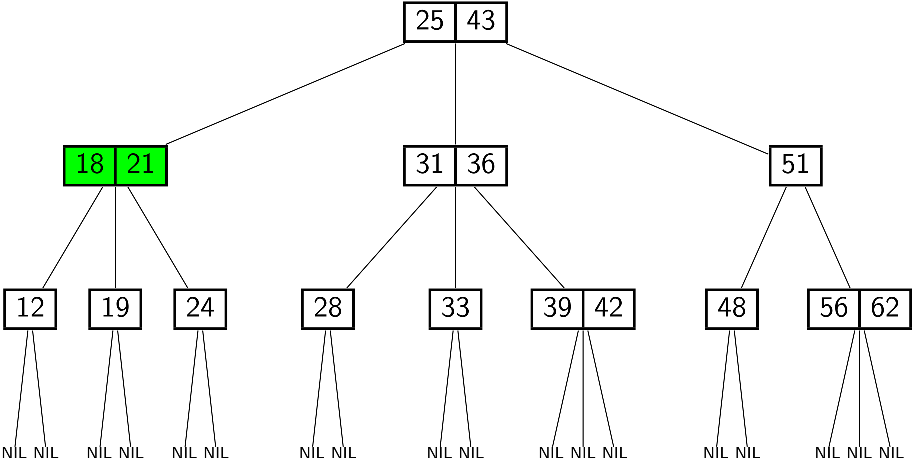
2-3 Tree Insertion

Example: *insert*(19)



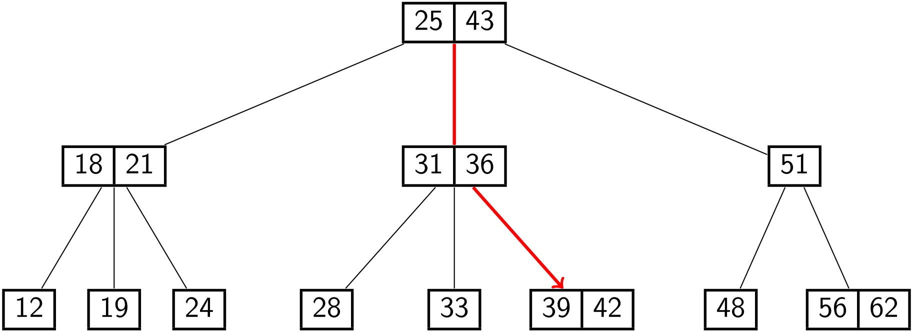
2-3 Tree Insertion

Example: *insert*(19)


25 43
18 21
31 36
51
12
19
24
28
33
39 42
48
56 62
NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL

2-3 Tree Insertion

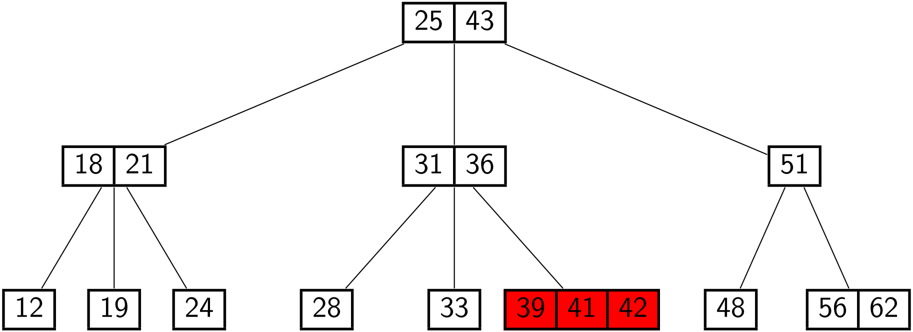
Example: *insert*(41)



(NIL-leaves not shown to simplify picture)

2-3 Tree Insertion

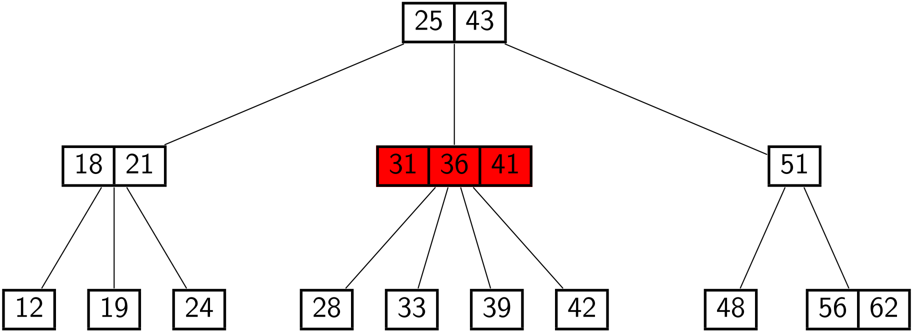
Example: *insert*(41)



(NIL-leaves not shown to simplify picture)

2-3 Tree Insertion

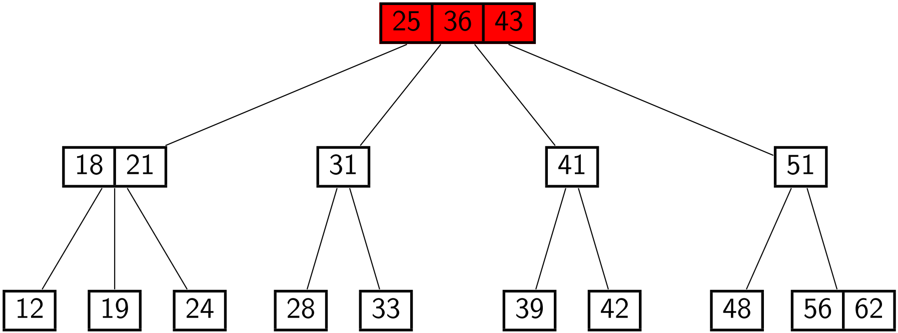
Example: *insert*(41)



(NIL-leaves not shown to simplify picture)

2-3 Tree Insertion

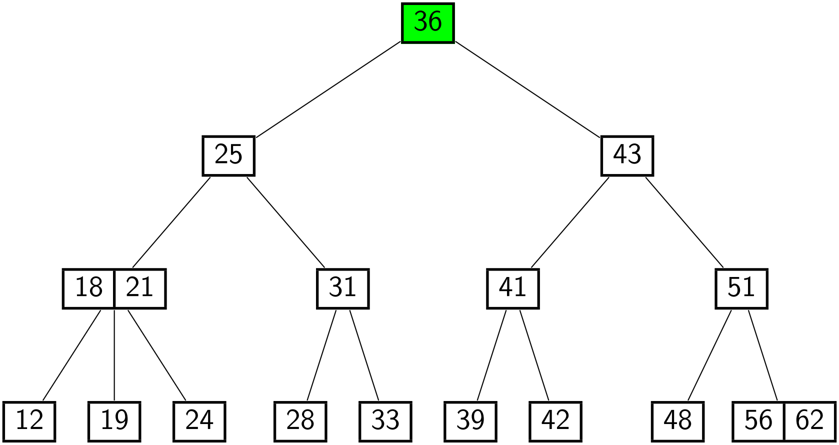
Example: *insert*(41)



(NIL-leaves not shown to simplify picture)

2-3 Tree Insertion

Example: *insert*(41)



(NIL-leaves not shown to simplify picture)

Deletion from a 2-3 Tree

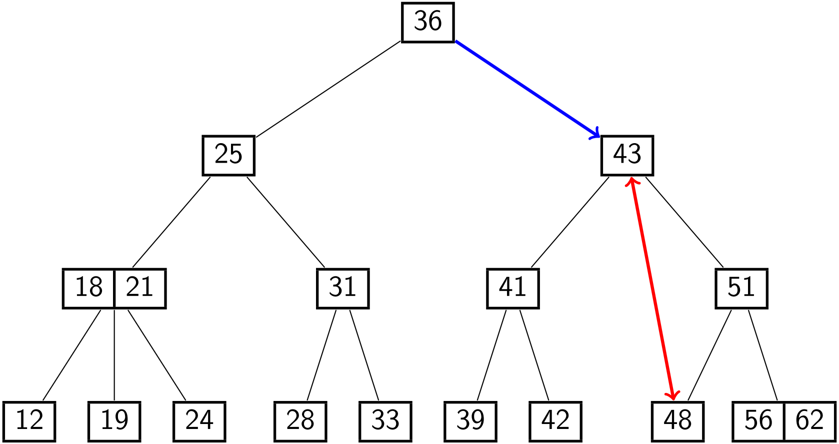
As with BSTs and AVL trees, we first swap the KVP with its successor, so that we always delete from a leaf.

Say we're deleting KVP x from a node V:

- If V is a 2-node, just delete x.
- ElseIf V has a 2-node *immediate* sibling U, perform a *transfer*:
  Put the "intermediate" KVP in the parent between V and U into V, and replace it with the adjacent KVP from U.
- Otherwise, we *merge* V and a 1-node sibling U:
  Remove V and (recursively) delete the "intermediate" KVP from the parent, adding it to U.

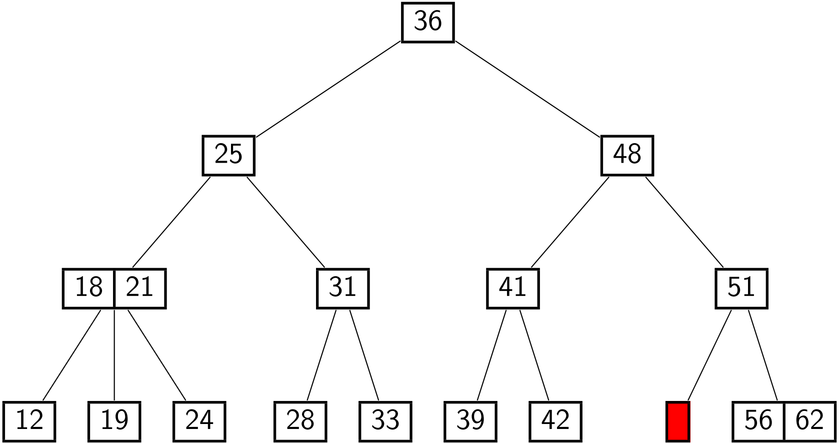
2-3 Tree Deletion

Example: *delete*(43)



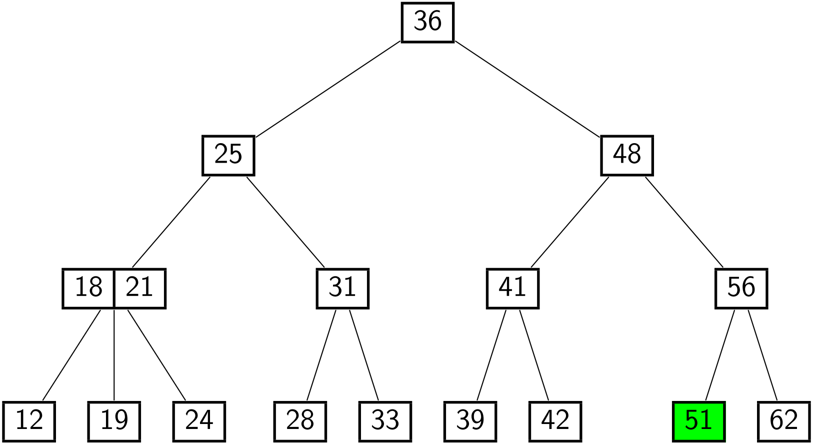
2-3 Tree Deletion

Example: *delete*(43)



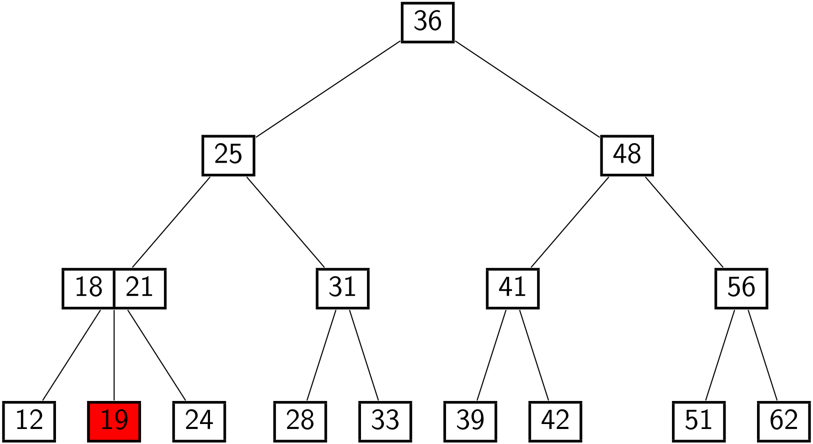
2-3 Tree Deletion

Example: *delete*(43)



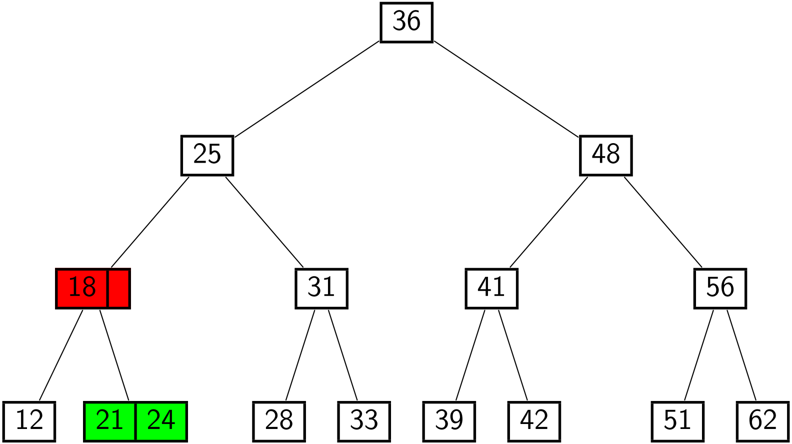
2-3 Tree Deletion

Example: *delete*(19)



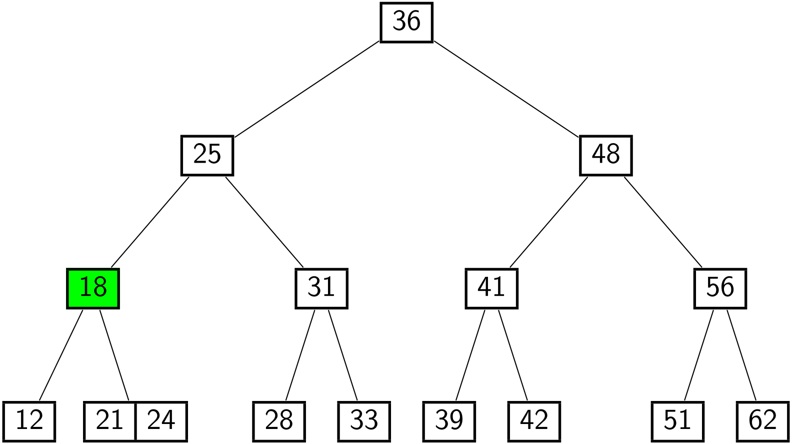
2-3 Tree Deletion

Example: *delete*(19)



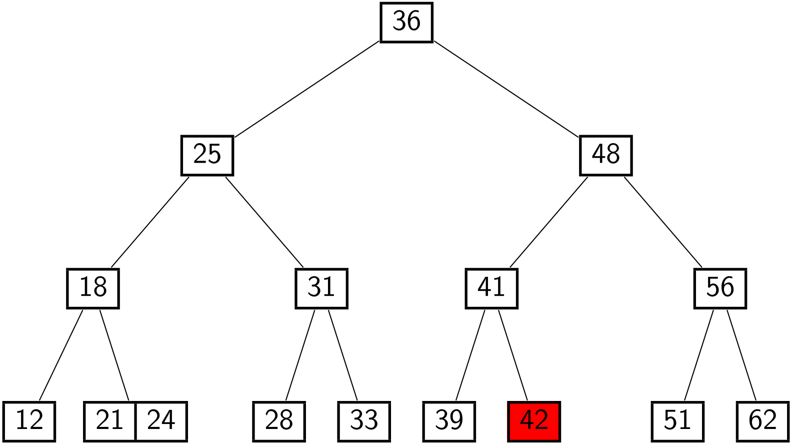
2-3 Tree Deletion

Example: *delete*(19)



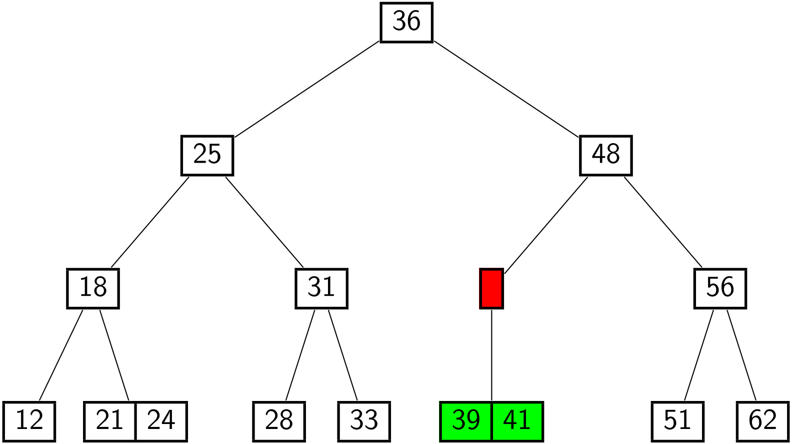
2-3 Tree Deletion

Example: *delete*(42)



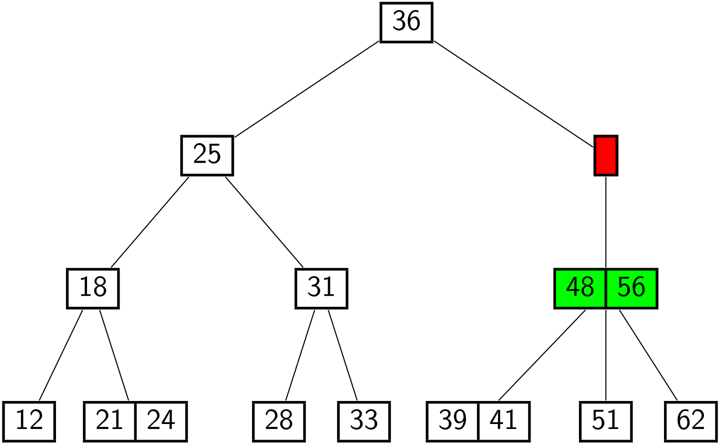
2-3 Tree Deletion

Example: *delete*(42)



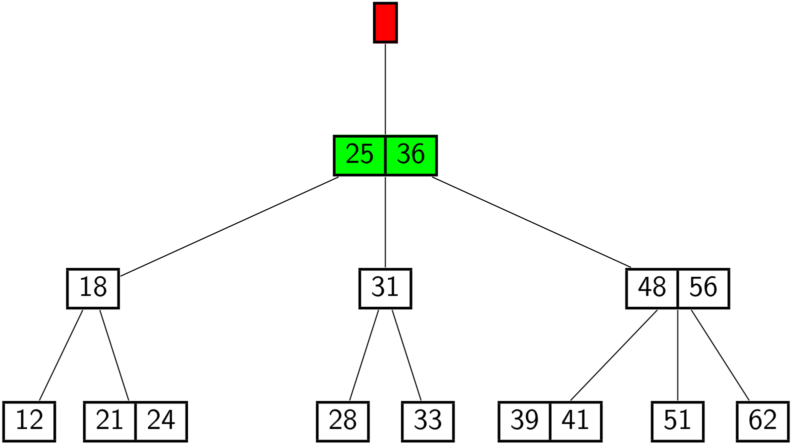
2-3 Tree Deletion

Example: *delete*(42)



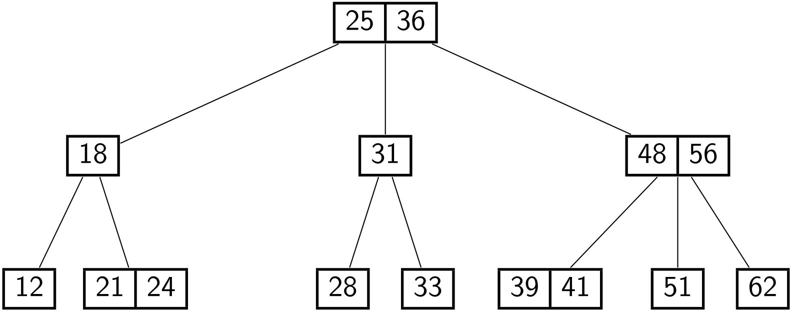
2-3 Tree Deletion

Example: *delete*(42)



2-3 Tree Deletion

Example: *delete*(42)



B-Trees

The 2-3 Tree is a specific type of (a, b)-tree:

An *(a, b)-tree of order M* is a search tree satisfying:

- Each internal node has at least a children, unless it is the root. The root has at least 2 children.
- Each internal node has at most b children.
- If a node has k children, then it stores $k-1$ key-value pairs (KVPs).
- Leaves store no keys and are at the same level.

for 3, except root node

A *B-tree of order M* is a $(\lceil M/2 \rceil, M)$-tree.
A 2-3 tree has $M = 3$.

search, *insert*, *delete* work just like for 2-3 trees.

Height of a B-tree

What is the least number of KVPs in a height-h B-tree?
(Height $=$ # levels **not** counting the dummy-level -1)

Level	Nodes	Links/node	KVP/node	KVPs on level
0	1	2	1	1
1	2	$M/2$	$M/2-1$	$2(M/2-1)$
2	$2(M/2)$	$M/2$	$M/2-1$	$2(M/2)(M/2-1)$
3	$2(M/2)^2$	$M/2$	$M/2-1$	$2(M/2)^2(M/2-1)$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
h	$2(M/2)^{h-1}$	$M/2$	$M/2-1$	$2(M/2)^{h-1}(M/2-1)$

$$\text{Total: } n \geq 1 + 2\sum_{i=0}^{h-1}(M/2)^i(M/2-1) = 2(M/2)^h - 1$$

Therefore height of tree with n nodes is $\Theta\big((\log n)/(\log M)\big)$.

Analysis of B-tree operations

Assume each node stores its KVPs and child-pointers in a dictionary that supports $O(\log M)$ search, insert, and delete.

Then *search*, *insert*, and *delete* work just like for 2-3 trees, and each require $\Theta(\mathit{height})$ node operations.

Total cost is $O\left(\frac{\log n}{\log M} \cdot (\log M)\right) = O(\log n)$.

Dictionaries in external memory

Recall: accessing a single location in *external memory* (e.g. hard disk) automatically loads a whole block (or "page").

In an AVL tree or 2-3 tree, $\Theta(\log n)$ pages are loaded in the worst case.

If M is small enough so an M-node fits into a single page,
then a B-tree of order M only loads $\Theta\big((\log n)/(\log M)\big)$ pages.

This can result in a *huge* savings:
memory access is often the largest time cost in a computation.

B-tree variations

Other strategies: *insert* and *delete* without *backtracking* via *pre-emptive splitting* and *pre-emptive merging*.

Red-black trees: Identical to a B-tree with minsize 1 and maxsize 3, but each 2-node or 3-node is represented by 2 or 3 binary nodes, and each node holds a "color" value of red or black.

B$^+$-trees: All KVPs are stored at the leaves
(interior nodes just have keys),
and the leaves are linked sequentially.

Hashing in External Memory

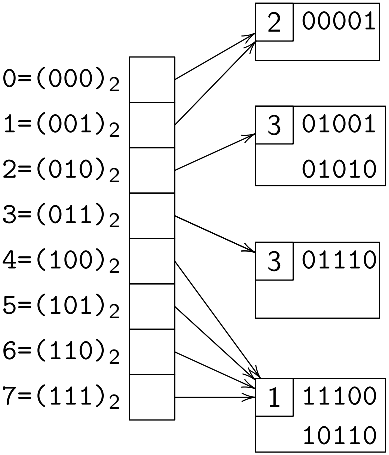
As before, if we have a *very large* dictionary that must be stored externally, how can we hash and minimize disk transfers?

Say external memory is stored in blocks (or "pages") of size S. Most hash strategies access many pages (data is scattered).

Exception: **Linear Probing**. All hash table accesses will usually be in the same page. But α must be kept small to avoid clustering, so there is a lot of wasted space.

New Idea: **Extendible Hashing**. Similar to a B-tree with height 1 and max size S at the leaves

Extendible Hashing Overview



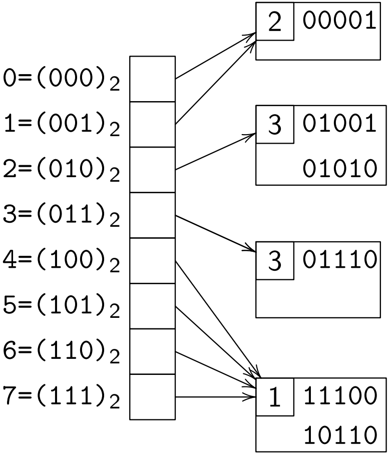
Assumption: Hash-function has values in $\{0, 1, \ldots, 2^L - 1\}$.

The *directory* (similar to root node) is stored in *internal memory*.
Contains array of size 2^d, where $d \leq L$ is called the *order*.

Each directory entry points to a *block* stored in *external memory*.
Each block contains at most S items.
(Many entries can point to the same block.)

To look up a key k in the directory, use the d leading bits of $h(k)$.

Extendible Hashing Details



Blocks are shared by entries in a specific manner:

- Every block B stores a *local depth* $k_B \leq d$.
- Hash values in B agree on leading k_B bits.
- All directory entries with the same k_B leading bits point to B.
- So 2^{d-k_B} directory entries point to block B.

Searching in extendible hashing

Searching is done in the directory, then in a block:

- Given a key k, compute $h(k)$.
- Leading d digits of $h(k)$ give index in directory.
- Load block B at this index into main memory.
- Perform a search in B for all items with hash value $h(k)$.
- Search among them for the one with key k.

Searching in extendible hashing

Searching is done in the directory, then in a block:

- Given a key k, compute $h(k)$.
- Leading d digits of $h(k)$ give index in directory.
- Load block B at this index into main memory.
- Perform a search in B for all items with hash value $h(k)$.
- Search among them for the one with key k.

Cost:

CPU time: depends on how the block are organized
(hash table, balanced tree, sorted array)

Disk transfers: 1 (directory resides in internal memory)

Insertion in Extendible Hashing

insert(k, v) is done as follows:

- Search for $h(k)$ to find the proper block B for insertion
- If the B has space, then put (k, v) there.

Insertion in Extendible Hashing

insert(k, v) is done as follows:

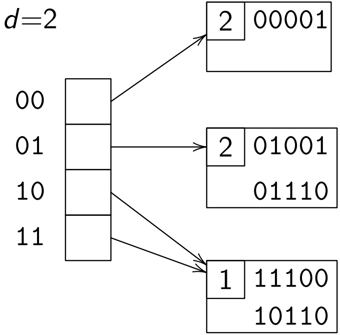
- Search for $h(k)$ to find the proper block B for insertion
- If the B has space, then put (k, v) there.
- ElseIf the block is full and $k_B < d$, perform a *block split*:
  - Split B into two blocks B_0 and B_1.
  - Separate items according to the $(k_B + 1)$-th bit.
  - Set local depth in B_0 and B_1 to $k_B + 1$
  - Update references in the directory
  - Try again to insert

Insertion in Extendible Hashing

insert(k, v) is done as follows:

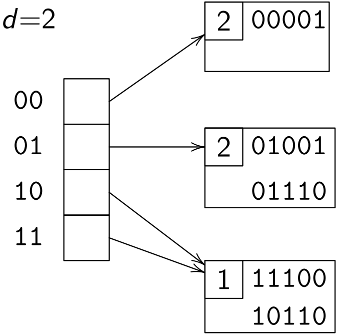
- Search for $h(k)$ to find the proper block B for insertion
- If the B has space, then put (k, v) there.
- ElseIf the block is full and $k_B < d$, perform a *block split*:
  - Split B into two blocks B_0 and B_1.
  - Separate items according to the $(k_B + 1)$-th bit.
  - Set local depth in B_0 and B_1 to $k_B + 1$
  - Update references in the directory
  - Try again to insert
- ElseIf the block is full and $k_B = d$, perform a *directory grow*:
  - Double the size of the directory $(d \leftarrow d + 1)$
  - Update references appropriately.
  - Then split block B (which is now possible).

Extendible hashing insert example with $S = 2$

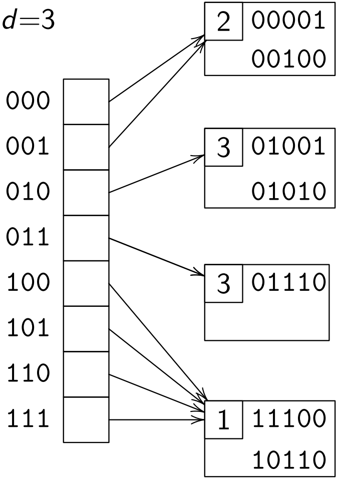


- `Insert( 00100 )`
- `Insert( 01010 )`

Extendible hashing insert example with $S = 2$



- `Insert( 00100 )`
- `Insert( 01010 )`



Extendible hashing conclusion

delete(k) is performed in a reverse manner to *insert*:

- Search for block B and remove k from it
- If block becomes too empty, then we perform a *block merge*
- If every block B has local depth $k_B \leq d - 1$, perform a *directory shrink*

But most likely just do *lazy deletion*.

Extendible hashing conclusion

delete(k) is performed in a reverse manner to *insert*:

- Search for block B and remove k from it
- If block becomes too empty, then we perform a *block merge*
- If every block B has local depth $k_B \leq d - 1$, perform a *directory shrink*

But most likely just do *lazy deletion*.

Cost of *insert* and *delete*:

CPU time: Search in a block depends on the implementation
$\Theta(S)$ to do/undo one split
Directory grow/shrink costs $\Theta(2^d)$ (but very rare).

Disk transfers: 1 when no split

Summary of extendible hashing

- Directory is much smaller than total number of stored keys and should fit in main memory.
- To make more space, we only add one block.
  Rarely do we have to change the size of the directory.
  Never do we have to move all items in the dictionary
  (in contrast to normal hashing).
- Space usage is not too inefficient: can be shown that under uniform hashing, each block is expected to be 69% full.
- Potentially extra CPU cost