CACHE-CONSCIOUS INDEXES

E0 261

Jayant Haritsa

Computer Science and Automation

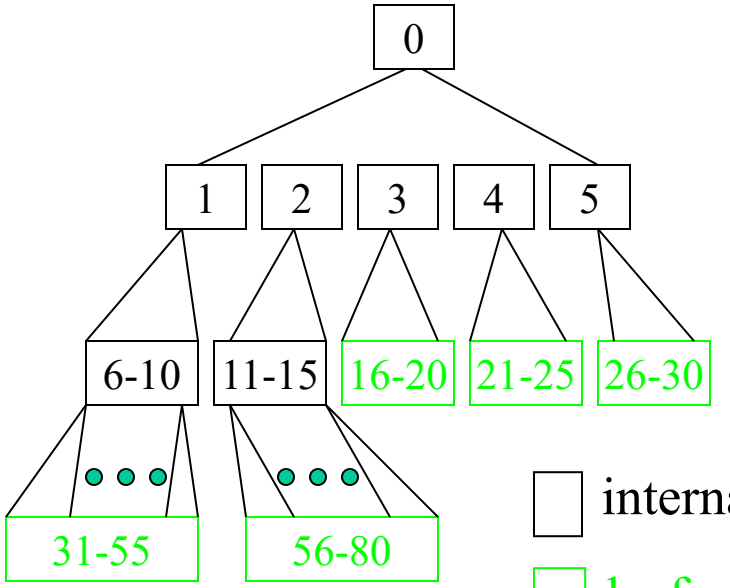
Indian Institute of Science

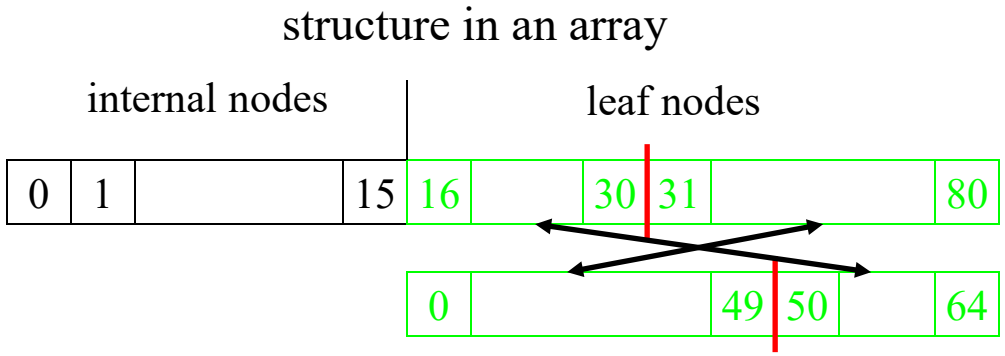
Prior Main Memory Indexes

- Binary search, T-Tree(balanced binary tree with several elements in a node) [1986]: minimize space overhead
  - not cache conscious, poor search
- Enhanced B+-tree [1983]: 100% filled, hard-code node search
  - more cache conscious, better search (node = cache line), but child pointers required
- Hashing: problem for ordered access, skewed data
- Cache-Sensitive Search (CSS) Tree: VLDB99

CSS-Trees

Basic Idea: Eliminate pointers by embedding a complete search tree into an array (directory array on top of sorted data array).





child nodes of node i : 5*i+1 to 5*i+5

internal node

leaf node

A CSS-Tree (m = 4)

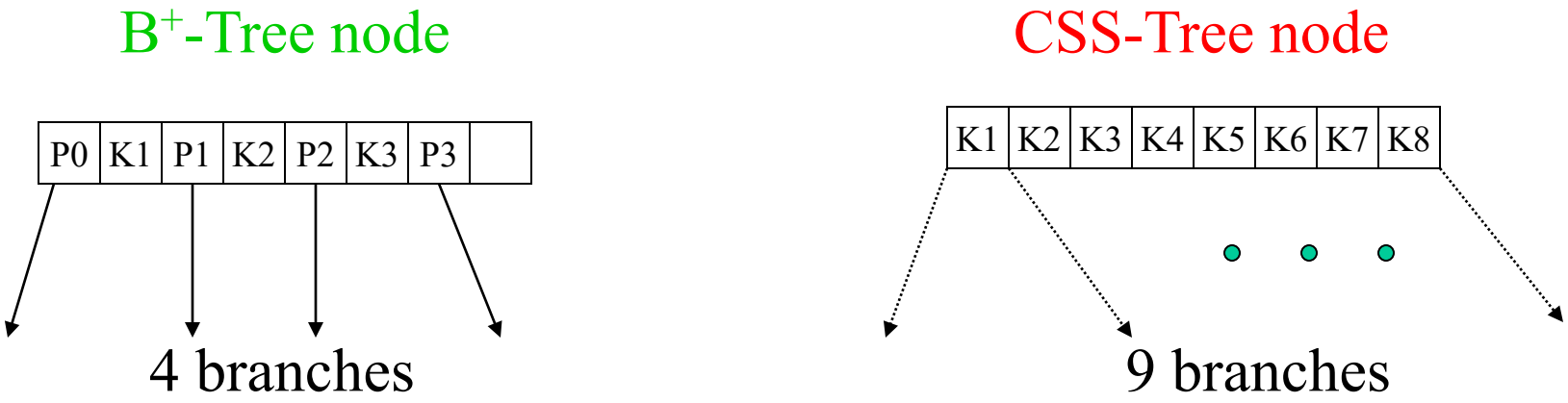
Tree Structure

- A complete (m+1)- ary search tree
- All nodes of tree are stored contiguously
- All nodes are by addressed by offset from start address of first node
- Children of internal node numbered b, are numbered from b(m+1)+1 to b(m+1)+m+1
- Key number i maps to node number $\lfloor i/m \rfloor$
  By this we can calculate address of any key

- Built bottom-up
- Updates require rebuilding of index

Utilization of a Cache line

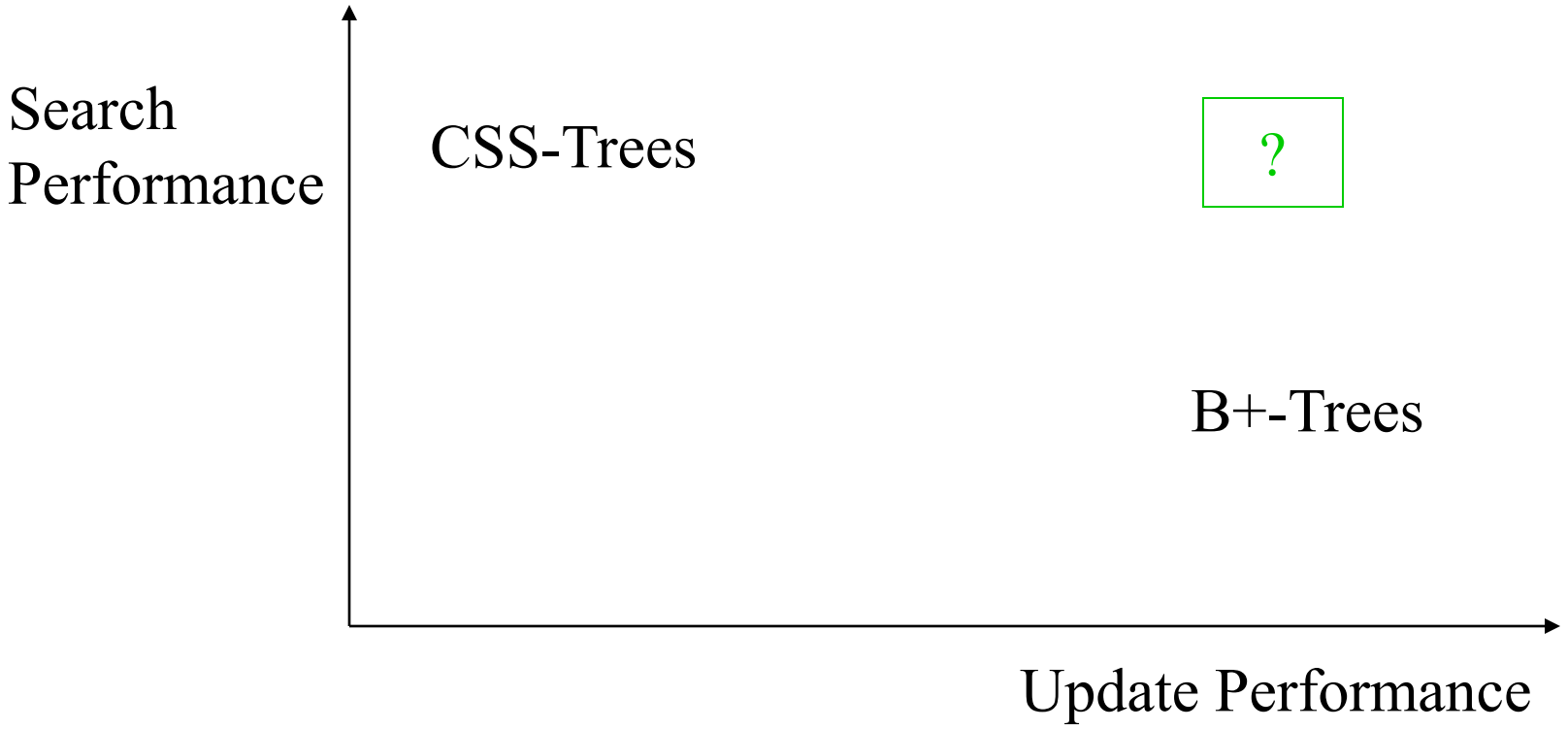
Cache line size = 32 bytes



Benefits for CSS-Trees:

- double number of keys per node
- fewer levels, less space
- search 20% faster

B+-Trees vs. CSS-Trees



Can we make B+-Trees more cache conscious?

CSB-TREES

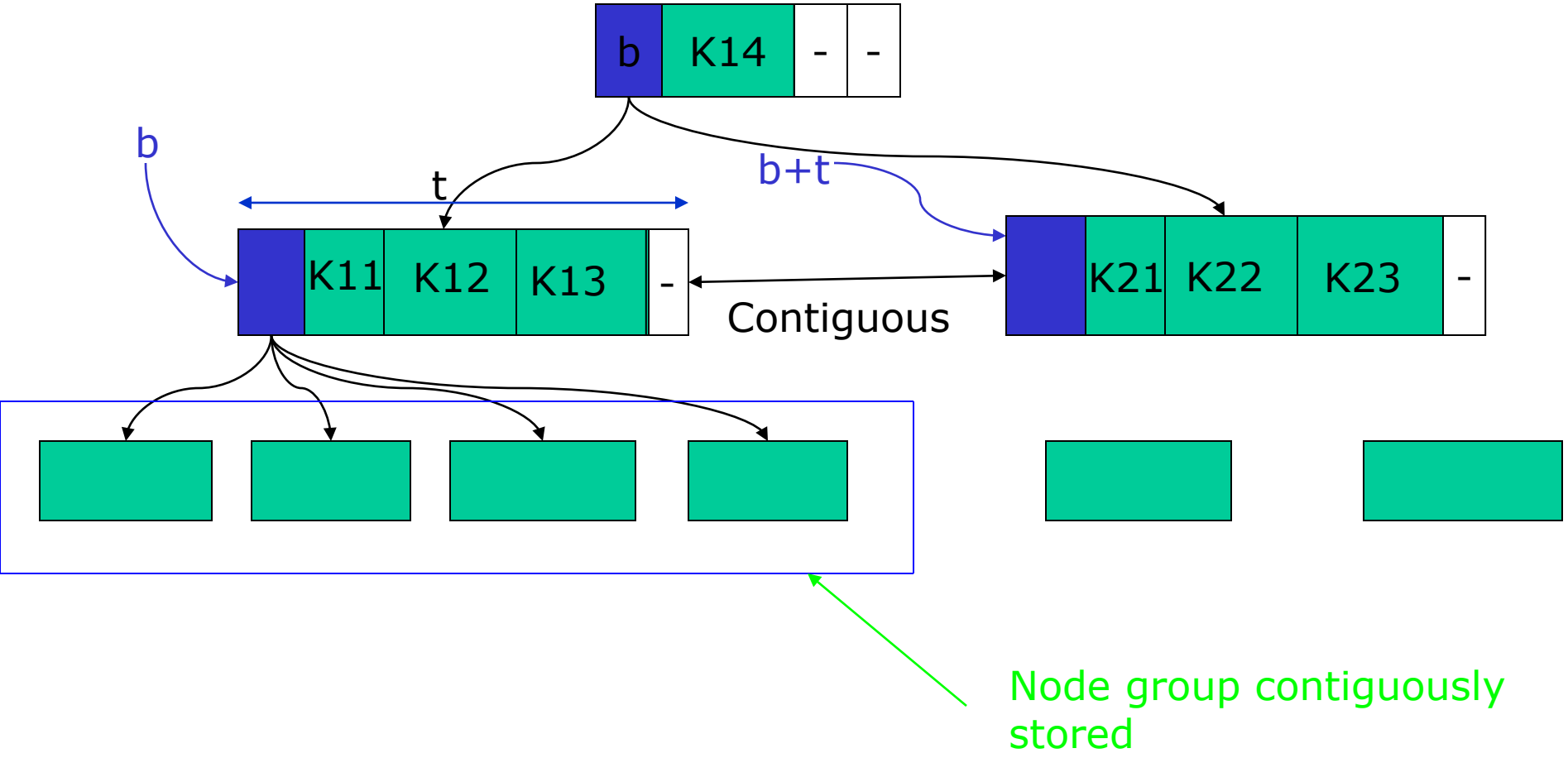
Tree Structure

- Balanced multiway search tree
- CSB^+ -Tree of order d contains m keys, where d<= m <= 2*d.
- All child nodes of a node in one node group.
- Node group is stored contiguously and therefore each node in the group can be accessed using an offset to the first node in the group
- Instead of m+1 pointers only one pointer to first child is needed $\Rightarrow$ partial pointer elimination technique $\Rightarrow$ more keys per node

Node Structure

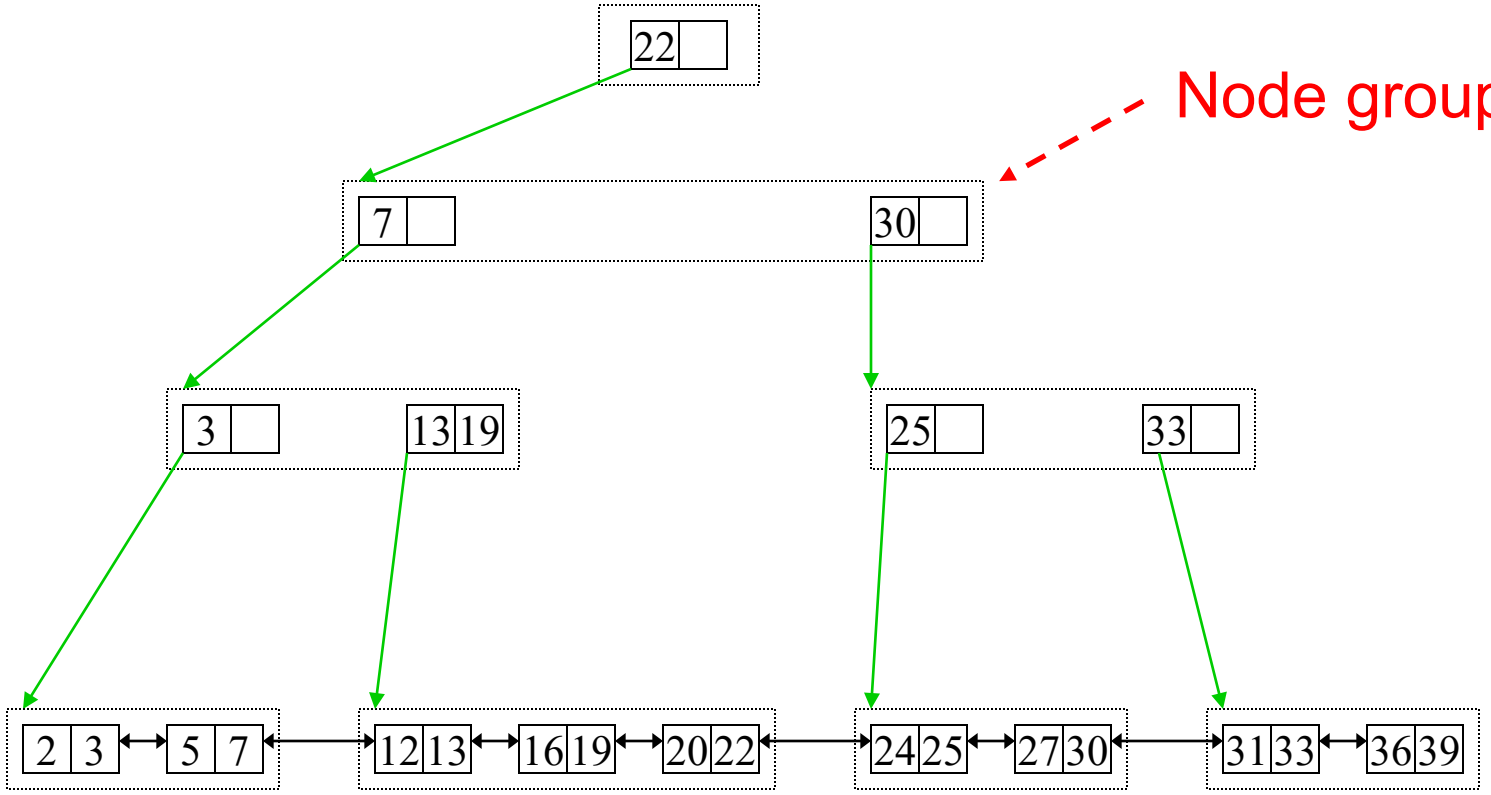
- Each node size = cache line
- Internal Node
  - nKeys: number of keys in the node
  - firstChild: pointer to first child node
  - keyList[2d]: list of keys
- Leaf Node
  - nKeys: number of keys in the node
  - Set of <key, tupleID> pairs
  - LSP / RSP: left/right sibling pointers

CSB+- tree example (Order 2)



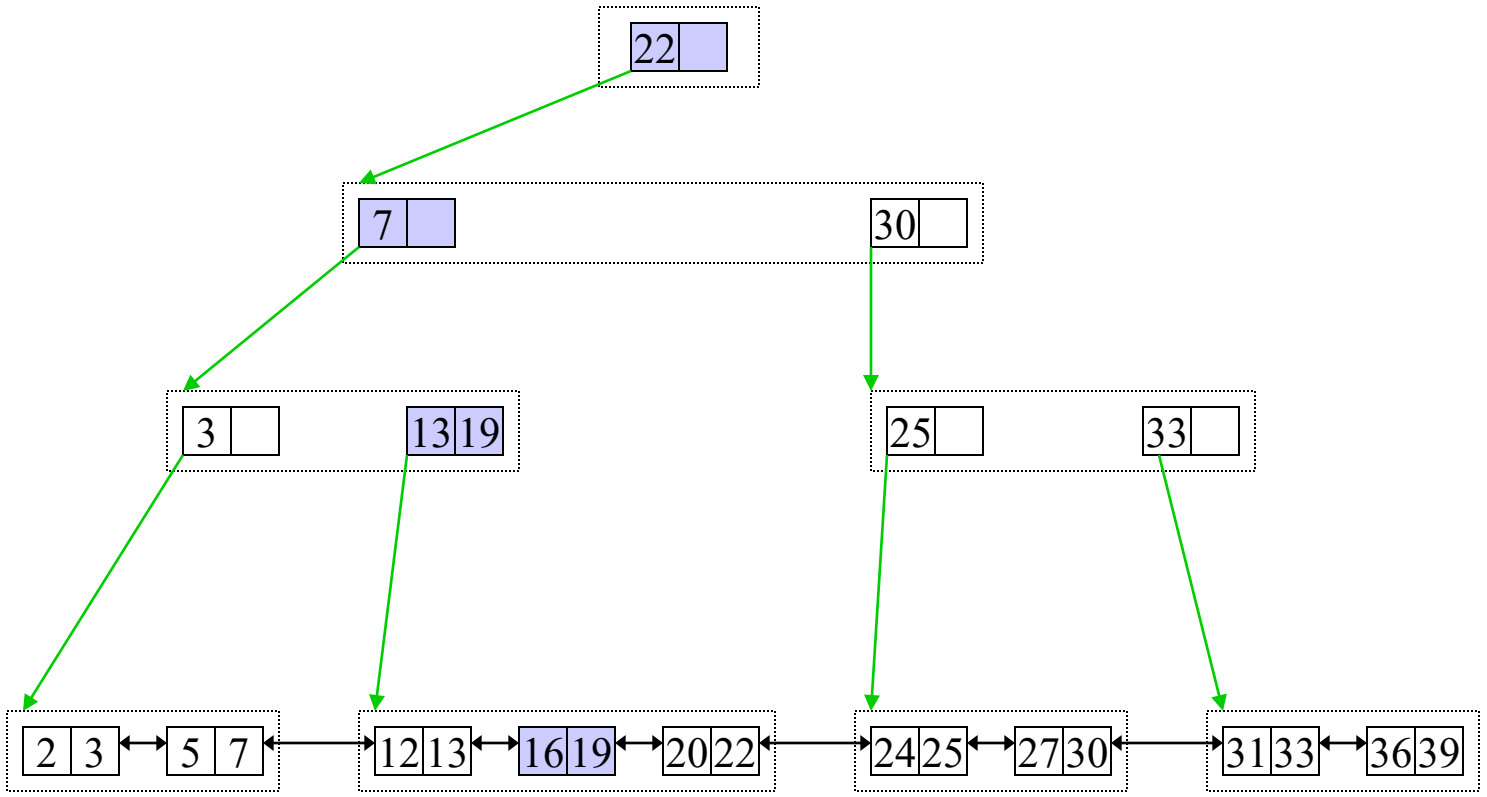
A CSB+-Tree of Order 1

[1,2] keys per node, node group max size = 3



Searching a CSB+-Tree

- Similar to B+ -Tree



Node Search Techniques

- Basic approach:
  - Binary search using standard while loop
- Code expansion approach:
  - Loop unfolding by using if-then-else
  - Pad unused slots with highest possible key $\Rightarrow$ no counter arithmetic
- Uniform approach:
  - Make the binary unfolding “right-deep”
  - Single traversal function
- Variable approach:
  - Traversal function dependent on key cardinality
  - Offline compute optimal traversal functions
  - Inline appropriate function at run time

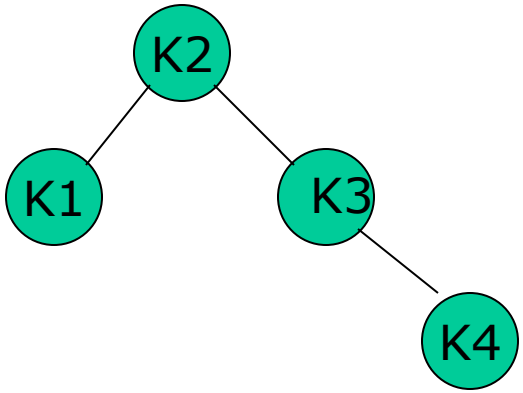
Example of Uniform approach

Search for value ‘a’

Keys K1,K2,K3,K4

```
if (a > K2)
 {
    if (a > K3)
      {
        if (a > K4)
            return p5;
        else
            return p4;
       }
    else
      return p3;
 }
 else
 {
    if (a > K1)
      return p2;
    else
       return p1;
 }
```

expanded code for
max key cardinality

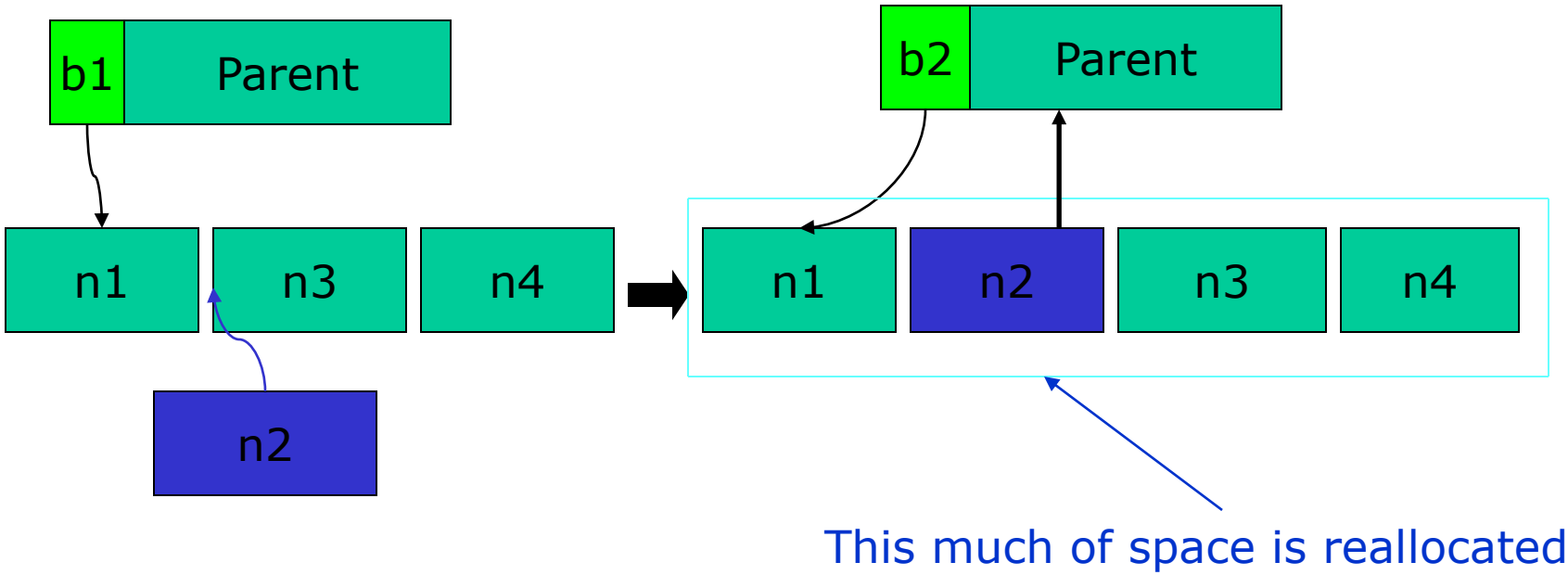


Larger

If node is not full, less chance that control goes inside it

Smaller

Insertion



Initially b1 was pointing to space where n1,n3,n4 are contiguously stored

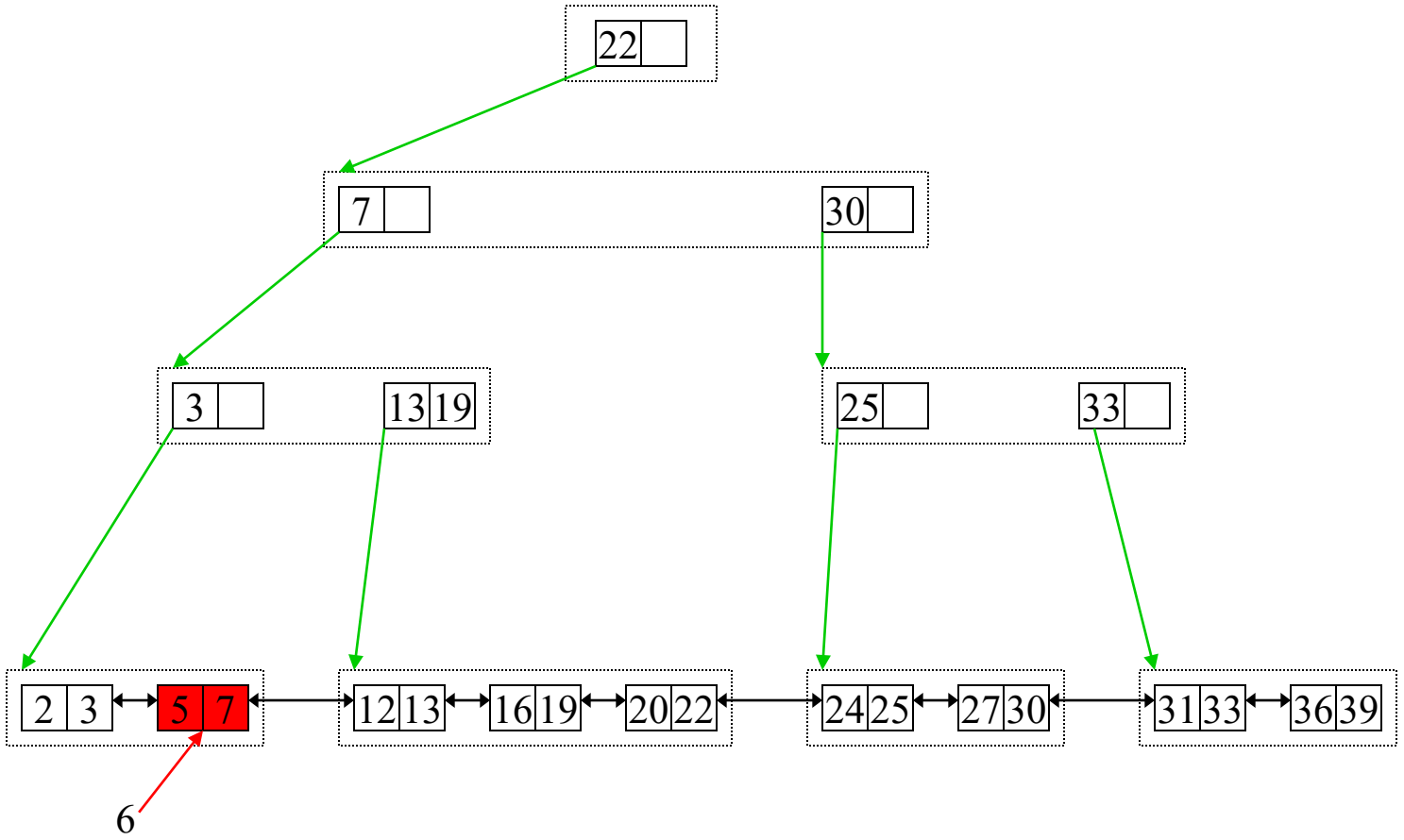
Now b2 is pointing to new allocated space where n1,n2 n3,n4 are contiguously stored

If the parent is full then parent will also spilt and process is repeated

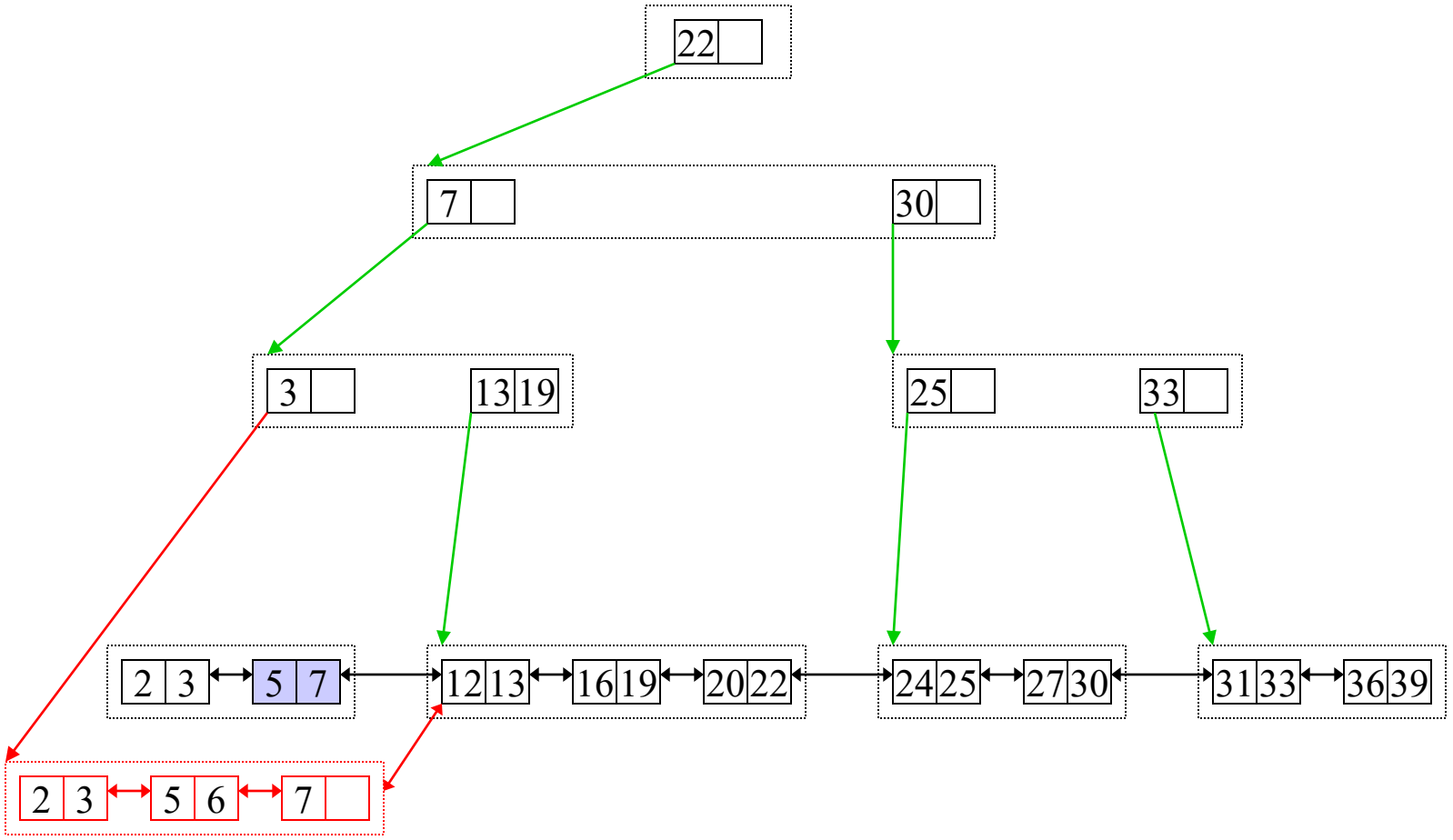
Insert Cost

- Split cost higher since whenever there is a new split, CSB have to create a new node group, whereas B-trees have to only create a new node

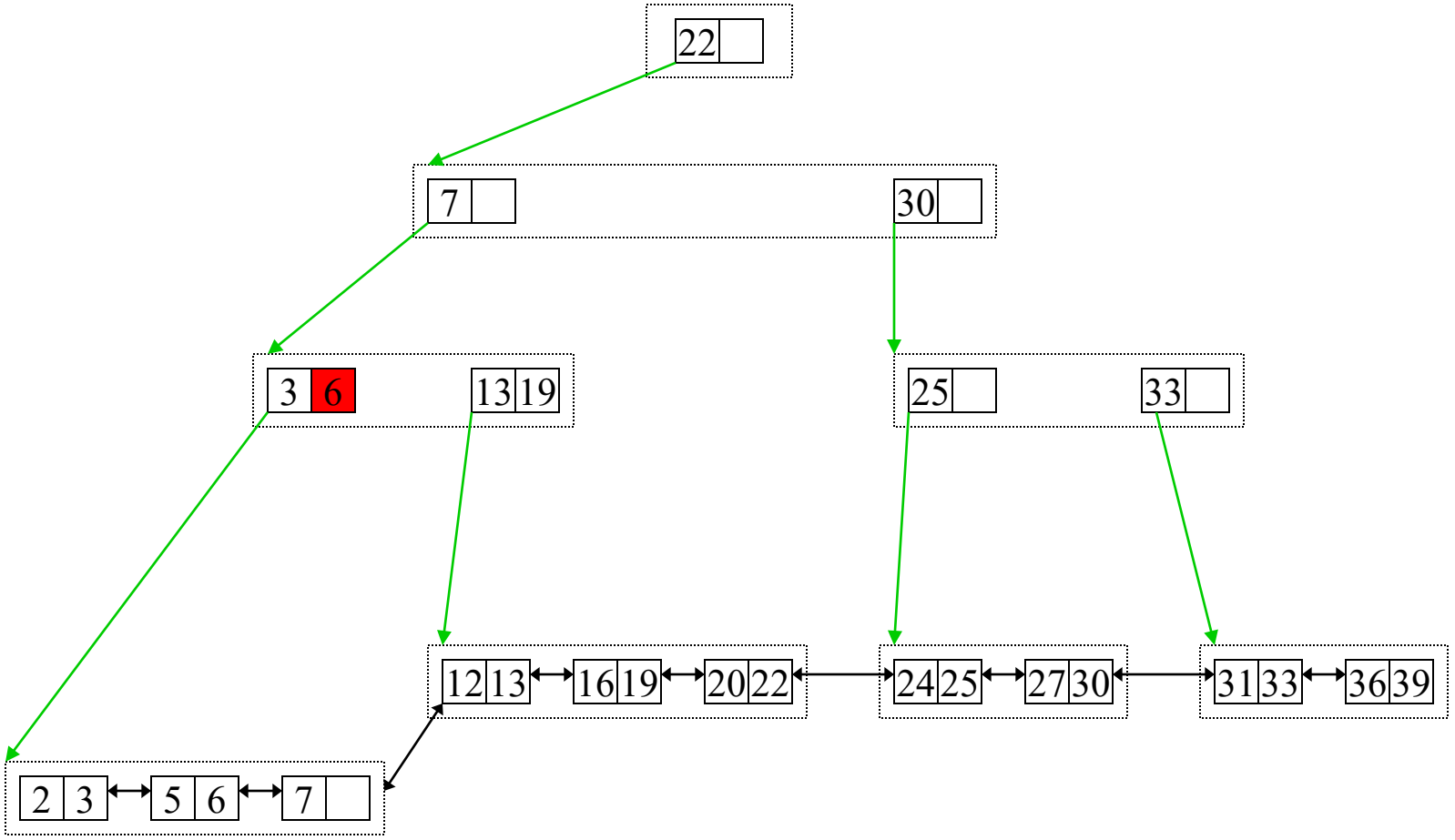
Example Insertion of 6



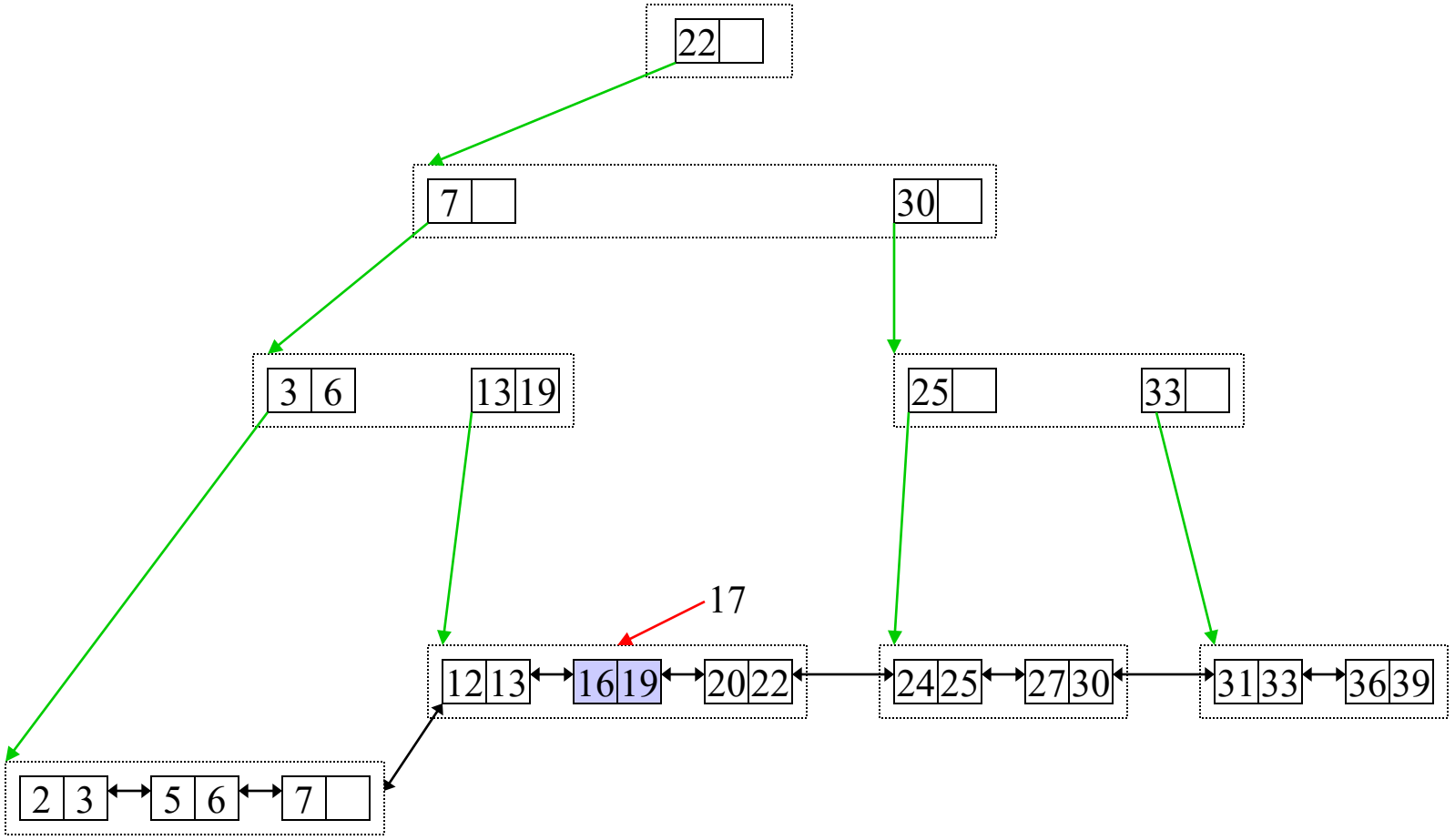
Insert 6 (contd)



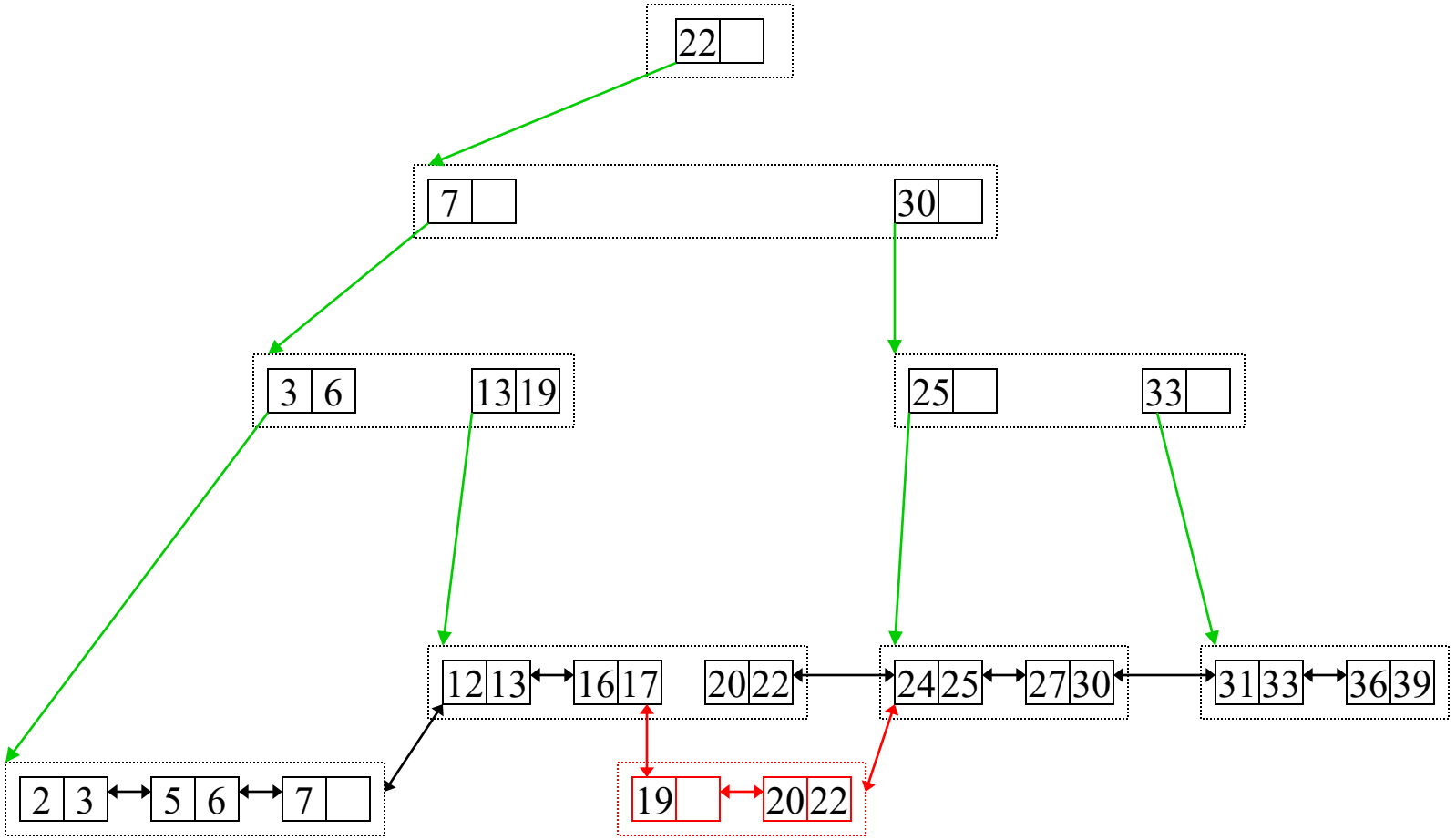
Insert 6 (contd)



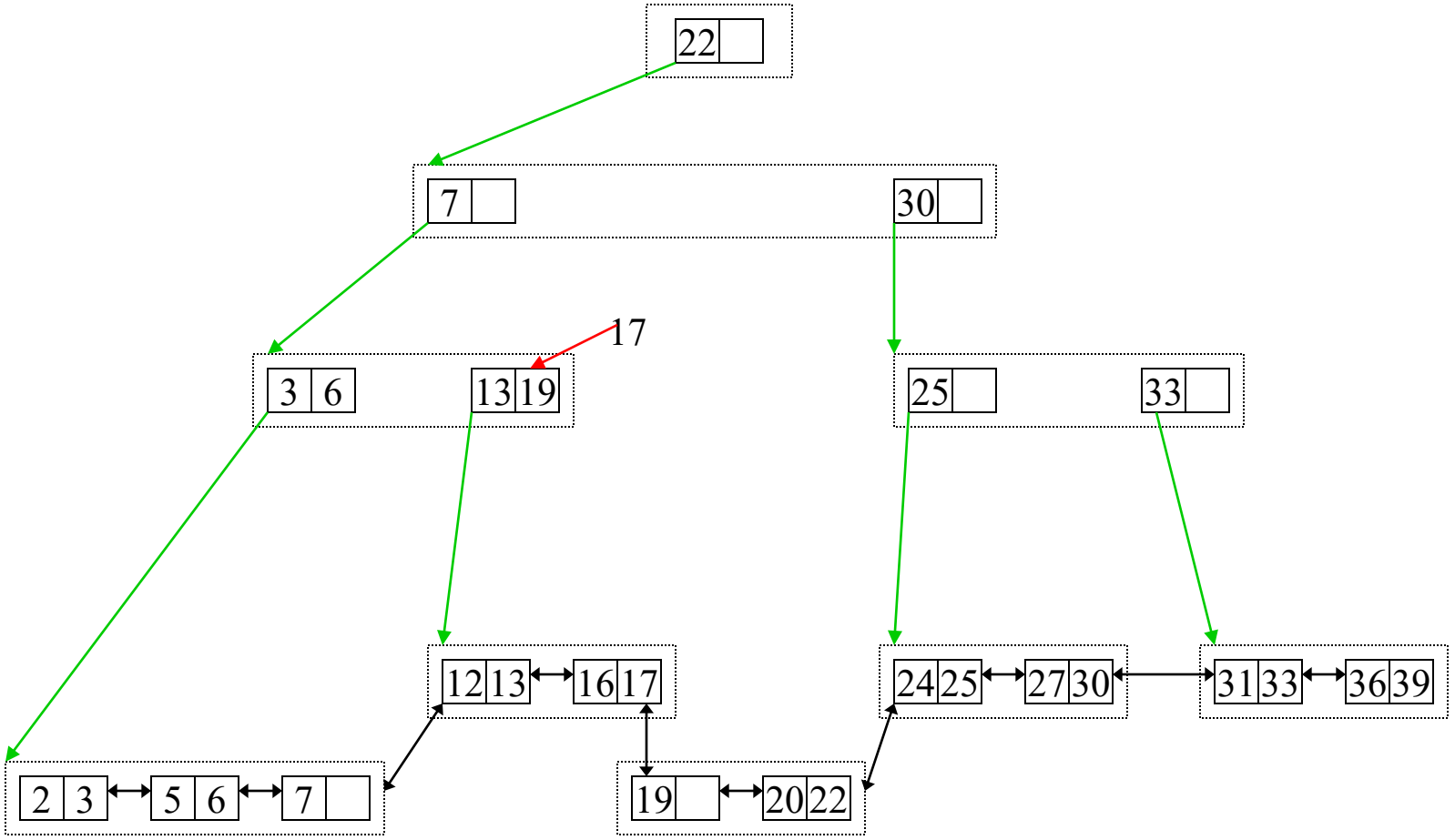
Insert 17


22
7
30
3 6
13 19
25
33
17
12 13
16 19
20 22
24 25
27 30
31 33
36 39
2 3
5 6
7

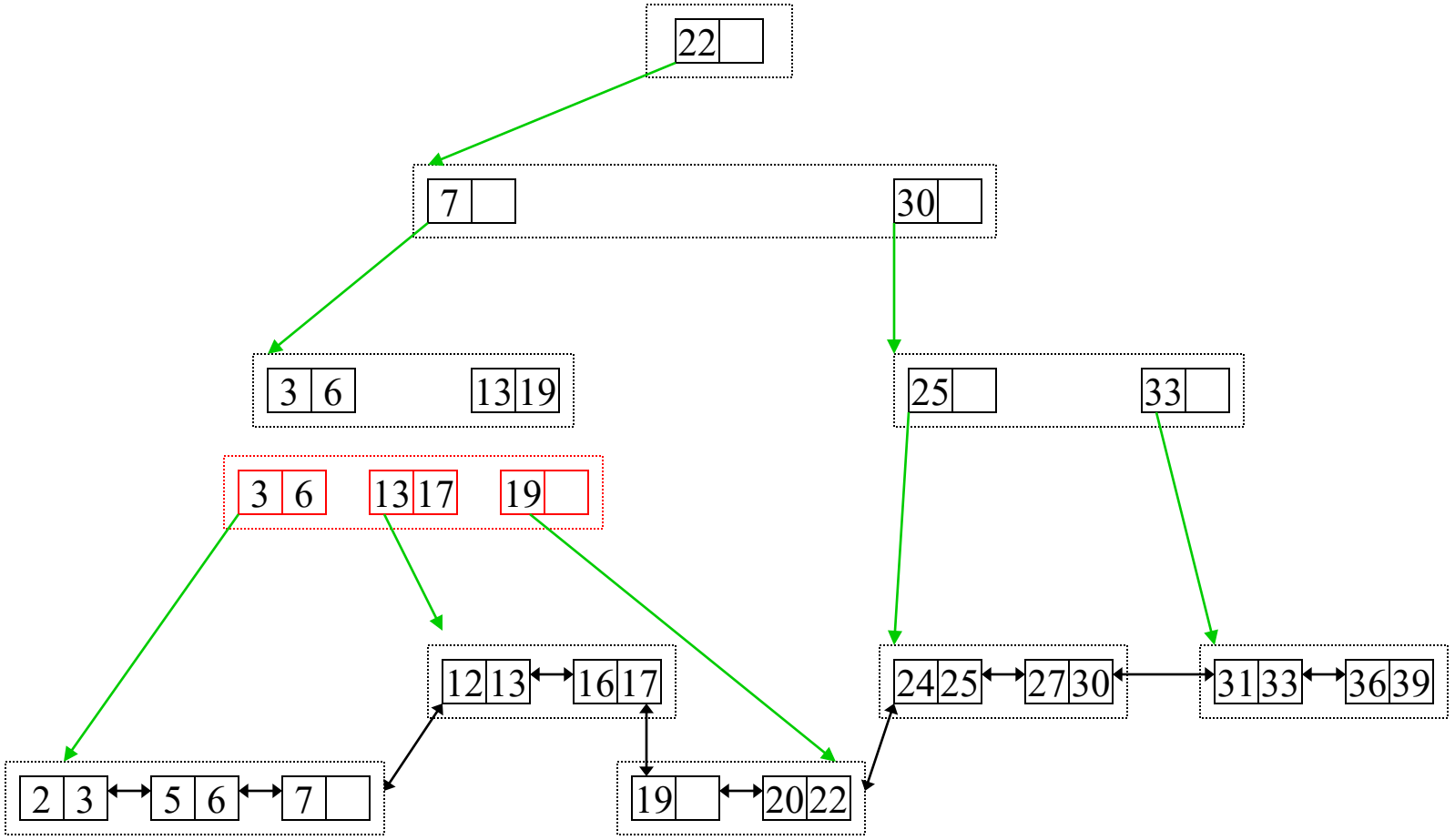
Insert 17 (contd)



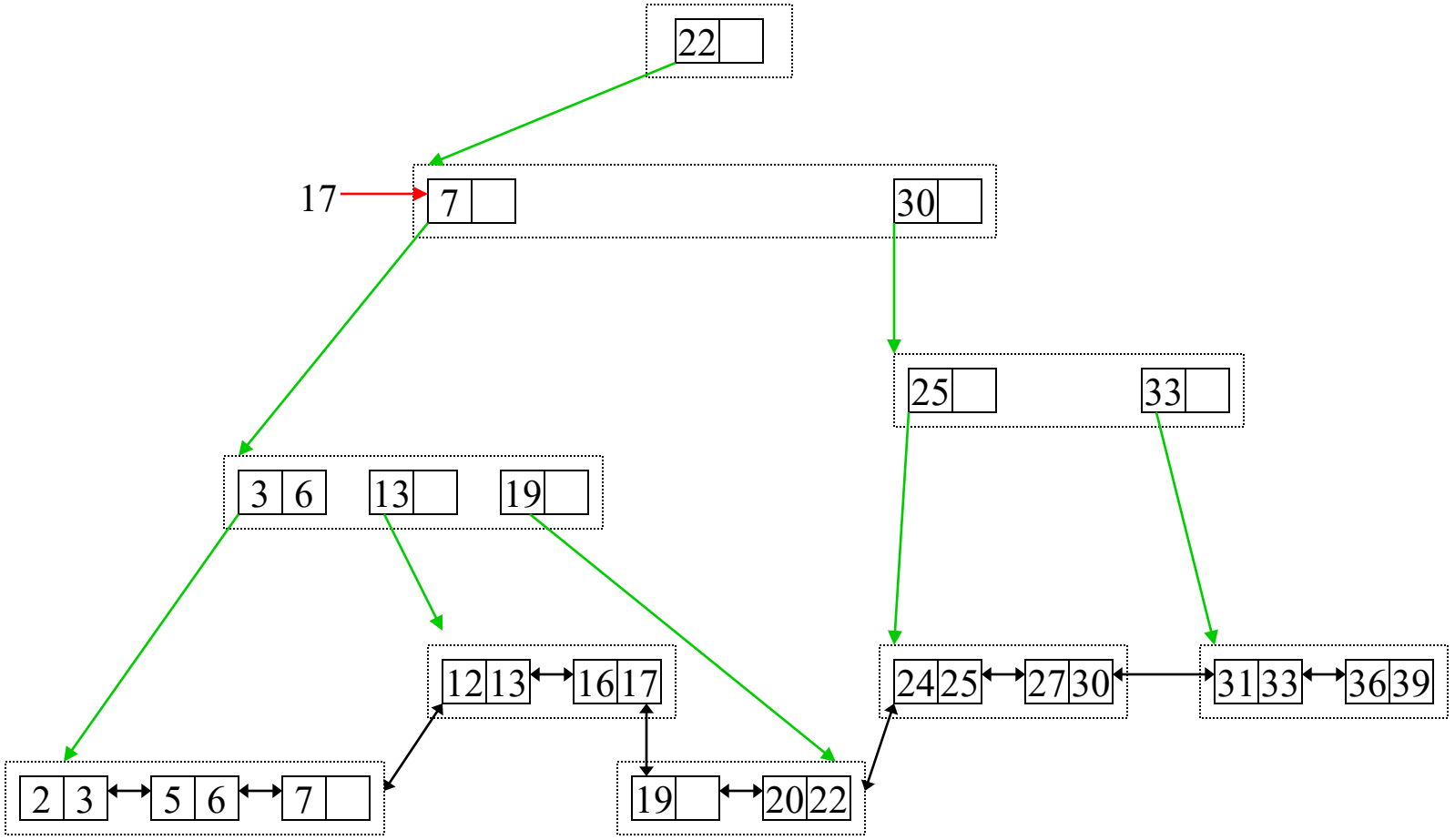
Insert 17 (contd)



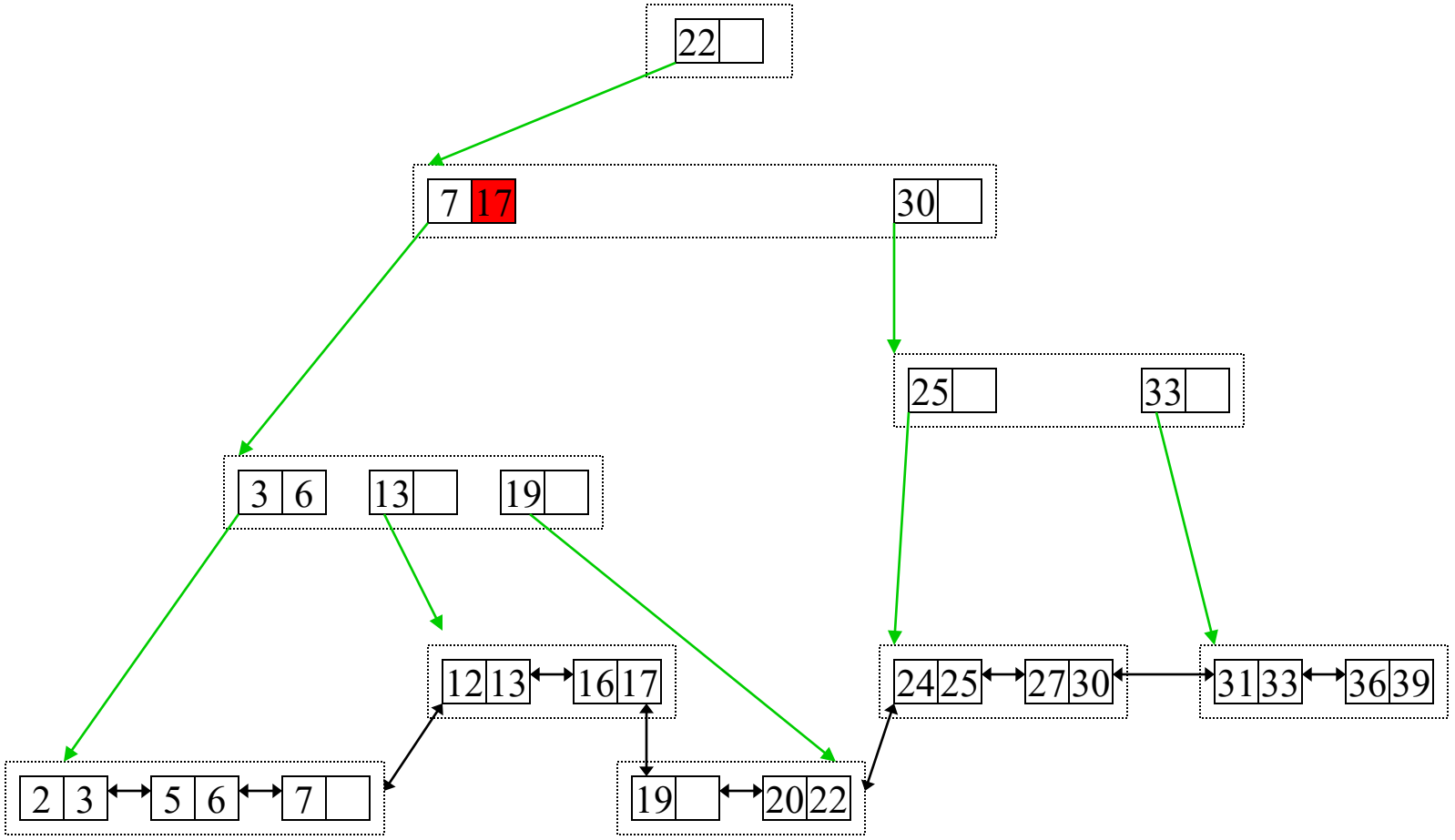
Insert 17 (contd)



Insert 17 (contd)



Insert 17 (contd)



Deletion

- Deletion is done lazily
- It may result in space underuse (less than 50%) but does not require adjustment of tree

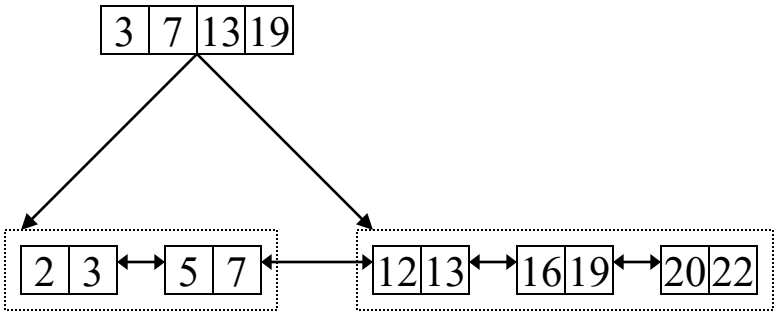
SEGMENTED CSB-TREES

Motivation

- As cache line sizes increase, impact of insertion cost is higher since node groups become proportionally larger
- Example:
  - cache line size of 128 bytes
    $\Rightarrow$ 30 Keys
    $\Rightarrow$ Node group size = 128 * (30+1) $\approx$ 4KB
    $\Rightarrow$ Copying of 4 KB for each split

Segmented CSB+-Trees

- divide child nodes into segments
- nodes in each segment are stored contiguously
- store pointers to each child segment in parent node



Benefit: copying less data on a split

Problem: Search becomes slower

FULL CSB-TREES

Full CSB+-Trees

- preallocate space for a full node group
- benefits:
  - cheaper copying on a split
    - move the nodes into the free space
  - less memory allocation overhead
    - triggered by node group overflow, not node overflow
- space overhead: 30%

PERFORMANCE

Benefit of CSB+-Trees

- Search is almost as efficient as CSS-Trees
  - n is no of leaf nodes, c is cache line size (64 bytes), m = no of slots in node

	Cache Misses	Typical Value (m=16)
B+-Tree	$\log_2 n/(\log_2 m - 1)$	$\log_2 n/3.0$
CSS-Tree	$\log_2 n/\log_2(m+1)$	$\log_2 n/4.1$
CSB+-Tree	$\log_2 n/\log_2(m-1)$	$\log_2 n/3.9$

- Incremental updates similar to Btrees
  - creating new node groups when split

Update Analysis

Update Cost = Search Cost + Split Cost

	Cache lines Accessed on a Split	Typical Values (in cache lines) for m = 16
B+-Tree	2	2
CSB+-Tree	(m-1)*2	30
CSB+-Tree (t seg)	(m-2t+1)*2/t	13
CSB+-Tree (full)	(m-1)/2	7.5

Feature Comparison

	Search	Update	Space	MM Overhead
B+-Tree	slower	faster	medium	medium
CSB+-Tree	faster	slower	lower	higher
CSB+-Tree (t seg)	medium	medium	lower	higher
CSB+-Tree (full)	faster	faster	higher	lower

If enough space, choose Full CSB+-Tree

If searches > updates (e.g., on-line shopping), choose variants of CSB+-Tree

Looking into the Future

- widening gap between CPU and memory speed
- operating system: 32-bit to 64-bit
- larger data set: longer search paths

Cache sensitive indexing structure even more useful in the future

END CC INDEXES

E0 261