Unit #10: B^{+}-Trees

CPSC 221: Algorithms and Data Structures

Lars Kotthoff[1]
larsko@cs.ubc.ca

[1]With material from Will Evans, Steve Wolfman, Alan Hu, Ed Knorr, and Kim Voll.

Unit Outline

- ▷ Minimizing disk I/O
- ▷ B^+-Tree properties
- ▷ Implementing B^+-Tree insert

Learning Goals

- Describe the structure, navigation and time complexity of a B^+-Tree.
- Insert keys into a B^+-Tree.
- Relate M, L, the number of nodes, and the height of a B^+-Tree.
- Compare and contrast B^+-Trees with other data structures.
- Justify why the number of I/Os becomes a more appropriate complexity measure (than the number of CPU operations) when dealing with large datasets and their indexing structures (e.g., B^+-Trees).
- Explain the difference between a B-Tree and a B^+-Tree.

Memory Hierarchy

Why worry about the number of disk I/Os?

Size		**Access Time**
hundreds of bytes	CPU registers	< 1 cycle
tens of kilobytes	Cache memory L1	a few cycles
	L2	
megabytes	L3	tens of cycles
gigabytes	Main memory	hundreds of cycles
terabytes	Disk	millions of cycles

Time Cost: Processor to Disk

Processor

- Operates at a few GHz (gigahertz = billion cycles per second).
- Several instructions per cycle.
- Average time per instruction $<$ 1ns (nanosecond = 10^{-9} seconds).

Disk

- Seek time $\approx$ 10ms (ms = millisecond = 10^{-3} seconds)
- (Solid State Drives have "seek time" $\approx$ 0.1ms.)

Result: 10 million instructions for each disk read!
Hold on… How long does it take to read a 1TB (Terabyte = 10^{12} bytes) disk? 1TB $\times$ 10ms = 10 billion seconds $>$ 300 years?
What's wrong? Each disk read/write moves more than a byte.

Memory Blocks

Each memory access to a slower level of the hierarchy fetches a block of data.

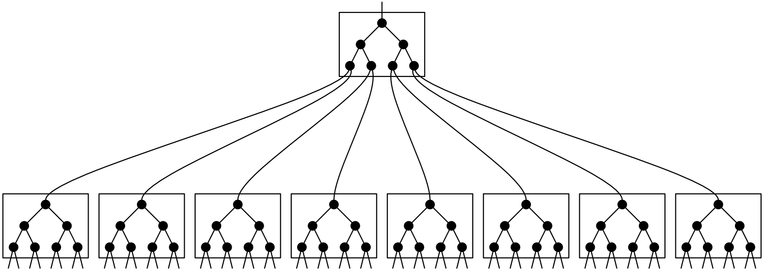
Block Size		**Block Name**
	CPU	
a few bytes	$\updownarrow$	word
	Cache	
10s bytes	$\updownarrow$	cache line
	Main memory	
a few kilobytes	$\updownarrow$	page
	Disk	

A block is the contents of consecutive memory locations.
So random access between levels of the hierarchy is very slow.

Chopping Trees into Blocks

Idea

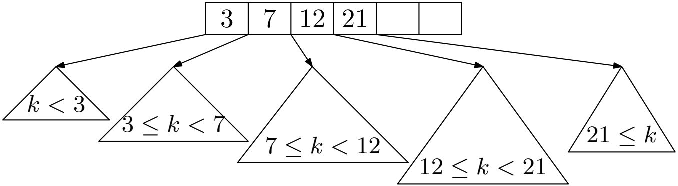
Store data for many adjacent nodes in consecutive memory locations.



Result

One memory block access provides keys to determine many (more than two) search directions.

M-ary Search Tree



M-ary tree property

- Each node has $\le M$ children

Result: Complete tree with n nodes has height $\Theta(\log_M n)$

Search tree property

- Each node has $\le M - 1$ search keys: $k_1 < k_2 < k_3 \ldots$
- All keys k in ith subtree obey $k_i \le k < k_{i+1}$ for $i = 0, 1, \ldots$.

Disk I/O (runtime) for `find`:

B^+-Trees

B^+-Trees of order M are specialized M-ary search trees:

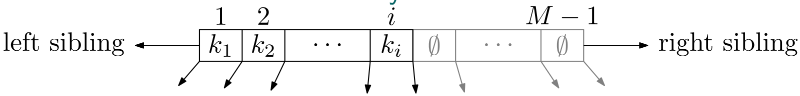
- ALL leaves are at the same depth!
- Internal nodes have between $\lceil M/2 \rceil$ and M children.
- Values are stored only at leaves. Search keys in internal nodes only direct traffic. B-Trees store (key, value) pairs at internal nodes.
- Leaves hold between $\lceil L/2 \rceil$ and L (key, value) pairs.
- The root is special. If internal, it has between 2 and M children. If a leaf, it holds at most L (key, value) pairs.

Result

- Height is $\Theta(\log_M n)$
- Insert, delete, find visit $\Theta(\log_M n)$ nodes
- M and L are chosen so that each node fills one page of memory. Each node visit (disk I/O) retrieves about $M/2$ to M keys or $L/2$ to L (key, value) pairs at a time.

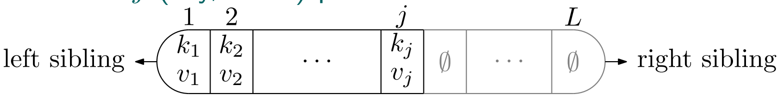
B^+-Tree Nodes

Internal node with i search keys



- $i+1$ subtree pointers
- parent and left & right sibling pointers

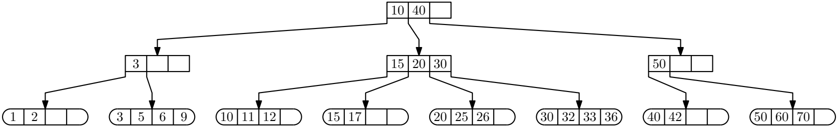
Leaf with j (key, value) pairs



- parent and left & right sibling pointers
- values may be pointers to disk records

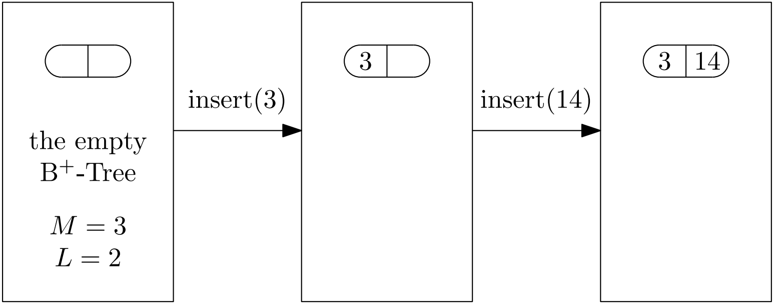
Each node may hold a different number of items.

Example B$^+$-Tree with $M = 4$ and $L = 4$



Values in leaf nodes are not shown.

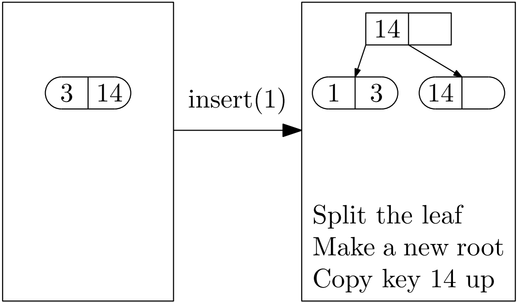
Making a B$^+$-Tree



The root is a leaf.

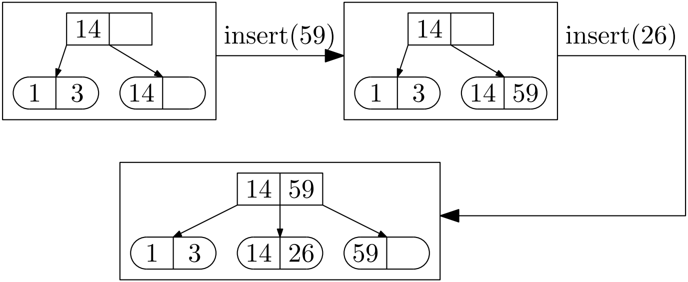
What happens when we now insert(1)?

Splitting the Root



Too many keys for one leaf!
So, make a new leaf and create a parent (the new root) for both.
Why are there duplicate 14 keys?

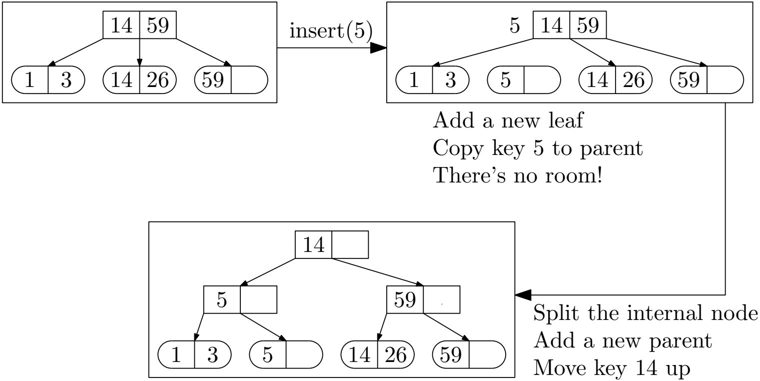
Splitting a Leaf



insert(26) causes too many keys for the (14 | 59) leaf.

So, make a new leaf and **copy** the middle key (the smallest key in the new leaf holding the larger keys) up to the common parent.

Propagating Splits



insert(5) causes too many keys for (1 | 3) leaf.

Copy up key 5 causes too many keys for $\boxed{14 \mid 59}$ node.

So, make a new internal node and **move up** the middle key.

Insertion Algorithm

1. Insert (key, value) pair in its leaf.
2. If the leaf now has $L + 1$ pairs: // overflow
   - ▷ Split the leaf into two leaves:
     - ▷ Original holds the $\lceil (L + 1)/2 \rceil$ small key pairs.
     - ▷ New one holds the $\lfloor (L + 1)/2 \rfloor$ large key pairs.
   - ▷ **Copy** smallest key in new leaf (the middle key) up to parent.
3. If an internal node now has M keys: // overflow
   - ▷ Split the node into two nodes:
     - ▷ Original holds the $\lceil (M - 1)/2 \rceil$ small keys.
     - ▷ New one holds the $\lfloor (M - 1)/2 \rfloor$ large keys.
   - ▷ If root, hang the new nodes under a new root. Done.
   - ▷ **Move** the remaining middle key up to parent & goto 3.