ECE430.217 Data Structures

B-Tree and B+Tree

Weiss Book Chapter 4.7

Byoungyoung Lee

https://compsec.snu.ac.kr

byoungyoung@snu.ac.kr

Lecture slides were prepared based on Brian Curless's notes

Lecture slides were prepared based on Douglas W. Harder's notes (**dwharder@alumni.uwaterloo.ca).**

Outline

Memory issues in designing data structures

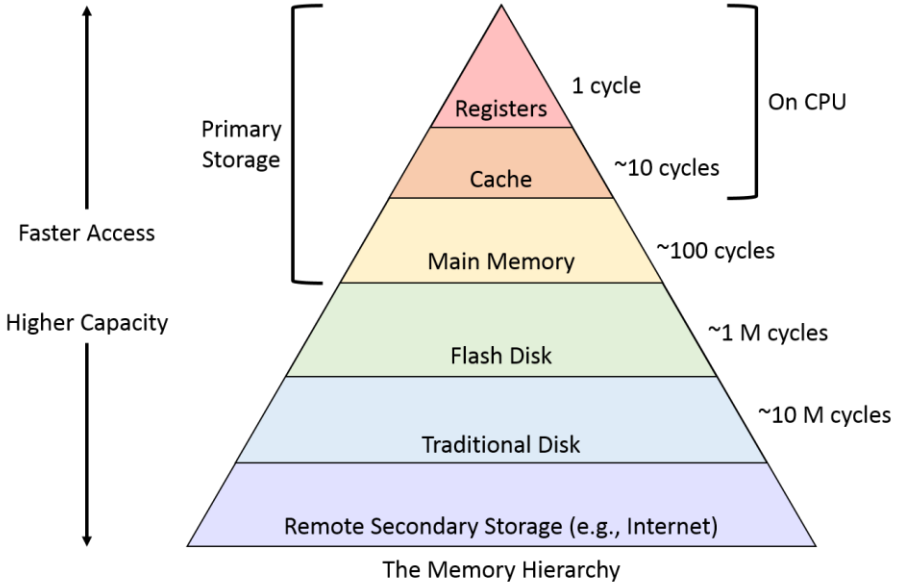
B-Tree

B+Tree

Memory Considerations

- When we discuss data structures, we never specifically mention where the data would be stored.

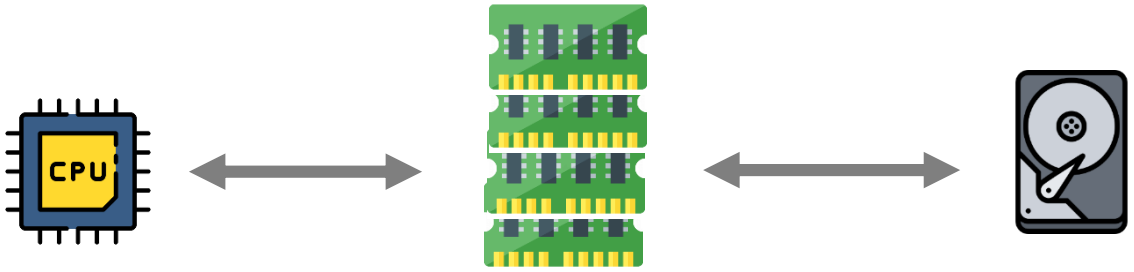
- In fact, memory hierarchy suggests that the design of data structures should be well aware of it



The Memory Hierarchy

Memory Considerations

- Memory things to think about when designing data structures
  - Access speed
  - Cost per memory size
  - The unit size of access
  - Stream access vs. Random access



CPU

Main memory (DRAM)

Disk

Fast access

Low capacity

Slow access

High capacity

Tree for very large datasets

- Suppose you got very many pieces of information
  - e.g., $n = 2^{30}$
  - Suppose each piece has 1KB data
  - Examples
    - Student records, where each piece holds each student's report
    - Sales history, where each piece holds sale records per item

- If you design the tree to store such data
  - The number of nodes: 2^{30}
  - Each node will occupy at least 1KB
  - Total? At least 1TB

```
1 class sales_node {
2   ID item_id;
3   RECORDS records; // 1KB
4   sales_node *p_left_tree;
5   sales_node *p_right_tree
6   // ...
7   // more
8   // ...
9 }
```

Issues: Tree for very large datasets

- Two performance issues of a “very large” tree
  - **Problem #1: Too many memory access**
    - Memory access is slow
    - But the number of memory access would be the height of the tree
    - Because you only load a single node per memory access
  - **Problem #2: Too big to store in the fast memory**
    - Can you store 1TB in the fast memory?
    - No, your main memory is (very likely) smaller than 1TB
    - So you will need to store the tree in the slow memory (i.e., a disk)

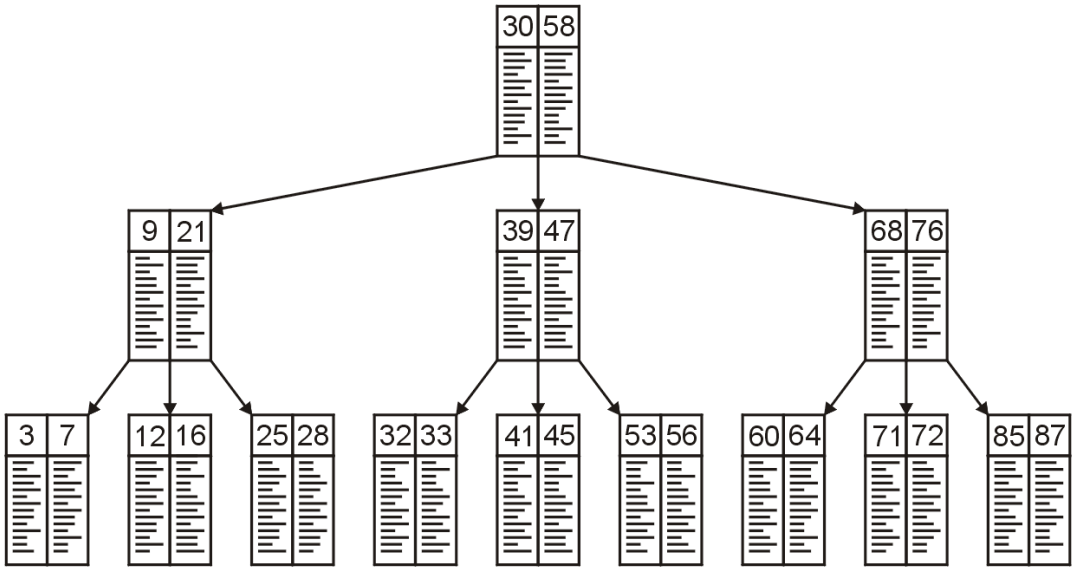
Ideas

- **Solution #1: Multiple keys per node**
  - Load multiple keys at once
  - Reduce the height of the tree
  - Trees with this feature
    - M-Way Search Trees, B-Tree, B+Tree

- **Solution #2: No data in the internal nodes**
  - Leaf nodes hold both keys/data, and internal nodes only hold keys
  - Entire “internal nodes” can be stored in the fast memory
  - Trees with this feature
    - B+Tree

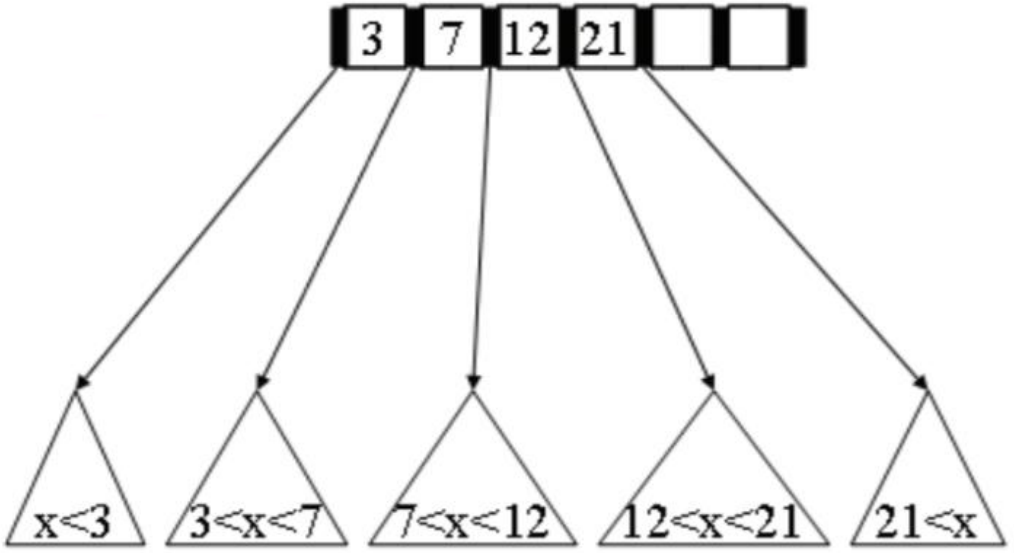
M-Way Search Trees

M-Way Search Trees: A search tree with maximum branching factor M



B-Trees

- Each node has keys up to M-1 keys
- Order property
  - Subtree between two keys x and y contain leaves with values v such that $x < v < y$

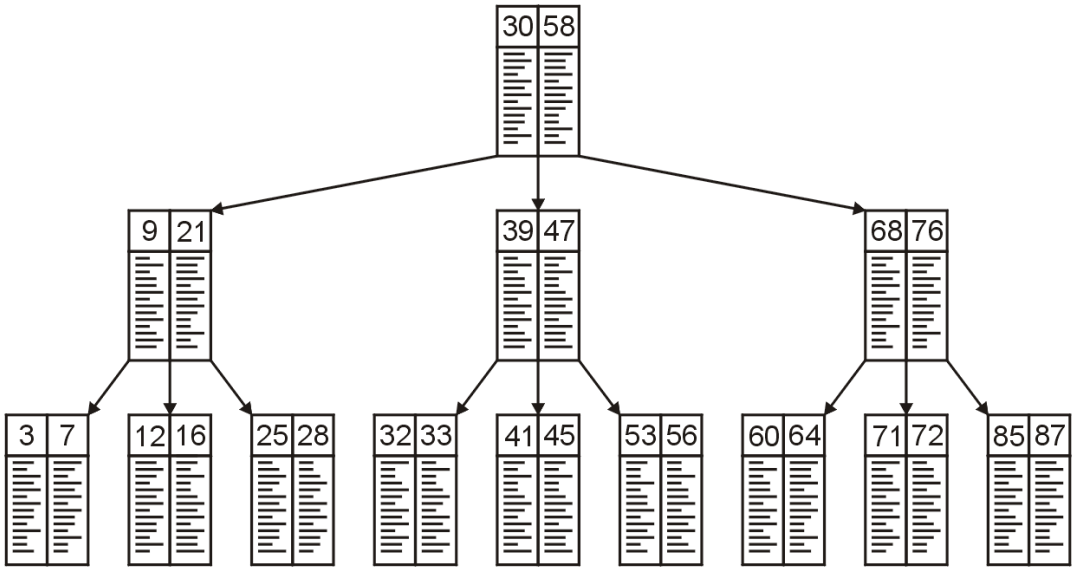


B-Tree Structure Property

- Root (special case)
  - Has between 2 and M children (or root could be a leaf node)
- Internal nodes
  - Store up to $M - 1$ keys
  - Have between $\lceil M/2 \rceil$ and M children
- Leaf nodes
  - Store between $\lceil (M - 1)/2 \rceil$ and $M - 1$ sorted keys
  - All at the same depth
- Note: Our B-Tree assignment has a slight different structural property, but the key ideas are the same

B-Tree: Example

- B-Tree with **M = 3**



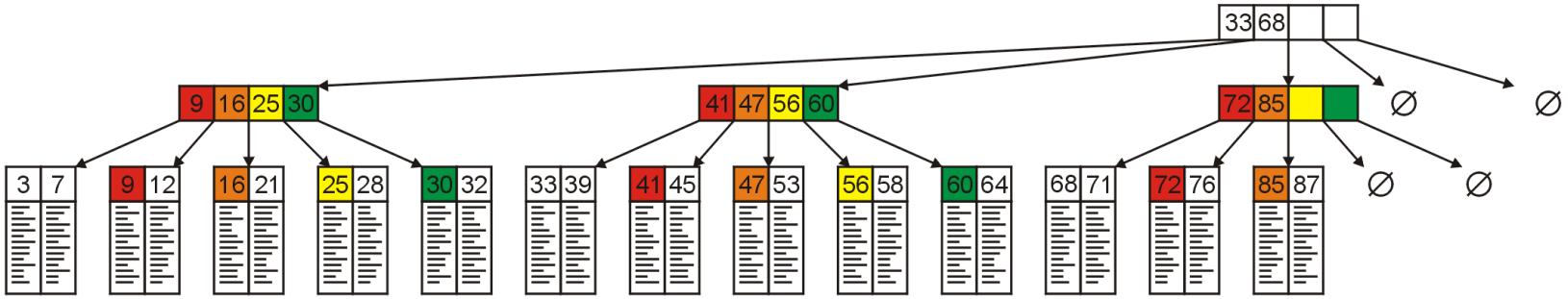
B+Trees

- Internal nodes have no data
  - Only leaf nodes have data
- Each internal node still has (up to) $M - 1$ keys
- Order property
  - Subtree between two keys x and y contain leaves with values v such that $x \leq v < y$
  - Note the symbol, '≤'
- Leaf nodes have up to L sorted keys

B+Tree Structure Property

- Root (special case)
  - Has between 2 and M children (or root could be a leaf node)
- Internal nodes
  - Store up to $M - 1$ keys
  - Have between $\lceil M/2 \rceil$ and M children
- Leaf nodes
  - Where data is stored
  - All at the same depth
  - Contain between $\lceil L/2 \rceil$ and L data items

B+Tree: Example



Disk Friendliness of B+Tree

- Many keys stored in a node
  - All brought to memory/cache in one disk access
- Internal nodes contain only keys
  - Only leaf nodes contain actual data
  - Much of tree structure can be loaded into memory irrespective of data object size
  - Data actually resides in disk

Performance Comparison: B+Tree vs. AVL Tree

- Suppose again you have $n = 2^{30}$ items
  - AVL Tree
    - Height: 43
  - B+Tree where M=256, L=256
    - Height: 4.3

- If you consider other factors, things are getting more interesting
  - The size of each item
  - The size of Cache
  - The size of DRAM
  - …

- We never talked about the costs to balance the tree though

Maintain the Balance of B+ Tree

- How to make B+ Tree balanced?
  - Insertion idea (bottom-up approach)
    - Step 1: Insert an item to the leaf
    - Step 2: If the node overflows, 1) split the node and 2) add the key to the parent
    - Step 3: If the parent overflows, go back to step 2
    - You may need to increase the height
  - Deletion idea (bottom-up approach)
    - Step 1: Remove an item from the leaf
    - Step 2: If the node underflows, 1) adopt from (or merge with) the neighbor and 2) update the parent
    - Step 3: If the parent underflows, go back to step 2
    - You may need to decrease the height

Applications

- Databases
  - Index structure for MySQL
  - A hash table can be a better option (will cover later)
- File systems
  - Apple’s HFS+, Microsoft NTFS, Linux’s EXT4 and btrfs
- Real-world challenges in designing and implementing trees
  - Parallel access: Multi-core processors are everywhere
  - Distributed storage: Too large to store in a single computing node
  - You will learn more from advanced courses: operating systems, computer architecture, database, etc.

References

Wikipedia, http://en.wikipedia.org/wiki/B+_tree

[1] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990, Ch. 19, p.381-99.

[2] Weiss, *Data Structures and Algorithm Analysis in C++, 3rd Ed.*, Addison Wesley, §4.7, p.159-64.

[3] Lecture slides by Brian Curless - https://courses.cs.washington.edu/courses/cse326/08sp/lectures/markup/11-b-trees-markup.pdf