ECE250:
Algorithms and Data Structures
B-Trees (Part A)

Ladan Tahvildari, PEng, SMIEEE
Associate Professor
Software Technologies Applied Research (STAR) Group
Dept. of Elect. & Comp. Eng.
University of Waterloo

Materials from CLRS: Chapter 18.1, 18.2
Acknowledgements

The following resources have been used to prepare materials for this course:

- MIT OpenCourseWare
- Introduction To Algorithms (CLRS Book)
- Data Structures and Algorithm Analysis in C++ (M. Wiess)
- Data Structures and Algorithms in C++ (M. Goodrich)

Thanks to many people for pointing out mistakes, providing suggestions, or helping to improve the quality of this course over the last ten years:

Disk Based Data Structures

- So far search trees were limited to main memory structures
  - Assumption: the dataset organized in a search tree fits in main memory (including the tree overhead)

- Counter-example: transaction data of a bank > 1 GB per day
  - Increase main memory (power failure?)
  - Use secondary storage media (hard disks, magnetic disks, etc.)

- Consequence: make a search tree structure secondary-storage-enabled
Algorithm Analysis

- The running time of disk-based algorithms is measured in terms of:
  - computing time (CPU)
  - number of disk accesses
    - sequential reads
    - random reads
Principles

- Pointers in data structures are no longer addresses in main memory

- If \( x \) is a pointer to an object
  - if \( x \) is in main memory \( \text{key}[x] \) refers to it
  - otherwise \( \text{DiskRead}(x) \) reads the object from disk into main memory (\( \text{DiskWrite}(x) \) – writes it back to disk)
Principles (2)

- A typical working pattern

```plaintext
01 ...
02 x ← a pointer to some object
03 DiskRead(x)
04 operations that access and/or modify x
05 DiskWrite(x) //omitted if nothing changed
06 other operations, only access no modify
07 ...
```
Binary-trees vs. B-trees

- Size of B-tree nodes is determined by the page size. One page – one node.
- A B-tree of height 2 containing > 1 Billion keys!
- Heights of Binary-tree and B-tree are logarithmic
  - B-tree: logarithm of base, e.g., 1000
  - Binary-tree: logarithm of base 2

---

1 node
1000 keys

1001 nodes,
1,001,000 keys

1,002,001 nodes,
1,002,001,000 keys
B-tree Definitions

- Node $x$ has fields
  - $n[x]$: the number of keys of that the node
  - $\text{key}_1[x] \leq \ldots \leq \text{key}_{n[x]}[x]$: the keys in ascending order
  - $\text{leaf}[x]$: true if leaf node, false if internal node
  - if internal node, then $c_1[x], \ldots, c_{n[x]+1}[x]$: pointers to children
  - leaf nodes have no children

- Keys separate the ranges of keys in the sub-trees. If $k_i$ is an arbitrary key in the subtree $c_i[x]$ then $k_i \leq \text{key}_i[x] \leq k_{i+1}$
B-tree Definitions (2)

- Every leaf has the same depth which is the tree’s height \( h \).
- In a B-tree of a degree \( t \):
  - Every node other than the root must have at least \( t-1 \) keys. Every internal node other than the root thus has at least \( t \) children.
  - Every node may contain at most \( 2t-1 \) keys. Therefore, an internal node may have at most \( 2t \) children.
  - The root node has between 0 and \( 2t \) children (i.e. between 0 and \( 2t-1 \) keys)
Height of a B-tree

- B-tree $T$ of height $h$, containing $n \geq 1$ keys and minimum degree $t \geq 2$, the following restriction on the height holds:

  $$ h \leq \log_t \frac{n + 1}{2} $$

- The diagram shows the structure of a B-tree with the following nodes and levels:

  - Level 0: 1 node
  - Level 1: 2 nodes
  - Level 2: 2t nodes

- The number of keys satisfies:

  $$ n \geq 1 + (t - 1) \sum_{i=1}^{h} 2t^{i-1} = 2t^h - 1 $$

Lecture 15
ECE250
B-tree Operations

- An implementation needs to support the following B-tree operations:
  - Searching (simple)
  - Creating an empty tree (trivial)
  - Insertion (complex)
  - Deletion (complex)
Searching

- Straightforward generalization of tree search (e.g., binary search trees)

```plaintext
BTreeSearch(x,k)
01 i ← 1
02 while i ≤ n[x] and k > key_i[x]
03    i ← i+1
04 if i ≤ n[x] and k = key_i[x] then
05    return (x,i)
06 if leaf[x] then
08    return NIL
09 else DiskRead(c_i[x])
10    return BTreeSearch(c_i[x],k)
```
Creating an Empty Tree

- Empty B-tree = create a root
- & write it to disk!

\[\text{BTreeCreate}(T)\]
01 \(x \leftarrow \text{AllocateNode}();\)
02 \(\text{leaf}[x] \leftarrow \text{TRUE};\)
03 \(n[x] \leftarrow 0;\)
04 \(\text{DiskWrite}(x);\)
05 \(\text{root}[T] \leftarrow x\)
B-tree Operations

An implementation needs to support the following B-tree operations

- **Searching** (simple)
- **Creating** an empty tree (trivial)
- **Insertion** (complex)
- **Deletion** (complex)
Splitting Nodes

- Nodes fill up and reach their maximum capacity $2t - 1$

- Before we can insert a new key, we have to “make room,” i.e., split nodes
Splitting Nodes (cont’) 

- Result: one key of $y$ moves up to parent + 2 nodes with $t-1$ keys.

```plaintext
y = c_i[x]
```

![Diagram](image-url)
Splitting Nodes (cont’)

```plaintext
BTreeSplitChild(x, i, y)
01 z ← AllocateNode()
02 leaf[z] ← leaf[y]
03 n[z] ← t-1
04 for j ← 1 to t-1
05    key_j[z] ← key_{j+t}[y]
06 if not leaf[y] then
07     for j ← 1 to t
08        c_{j}[z] ← c_{j+t}[y]
09     n[y] ← t-1
10 for j ← n[x]+1 downto i+1
11    c_{j+1}[x] ← c_{j}[x]
12    c_{i+1}[x] ← z
13 for j ← n[x] downto i
14    key_{j+1}[x] ← key_{j}[x]
15 key_{i}[x] ← key_{t}[y]
16 n[x] ← n[x]+1
17 DiskWrite(y)
18 DiskWrite(z)
19 DiskWrite(x)
```

x: parent node  
y: node to be split and child of x  
i: index in x  
z: new node

![Diagram](image-url)
Split: Running Time

- A local operation that does not traverse the tree
- $\Theta(t)$ CPU-time, since two loops run $t$ times
- 3 I/Os
Inserting Keys

- Done recursively, by starting from the root and recursively traversing down the tree to the leaf level.

- Before descending to a lower level in the tree, make sure that the node contains \(< 2t - 1\) keys.
Inserting Keys (cont’)

- Special case: root is full (BtreeInsert)

```
BTreeInsert(T, k)
01 r ← root[T]
02 if n[r] = 2t - 1 then
03   s ← AllocateNode()
04   root[T] ← s
05   leaf[s] ← FALSE
06   n[s] ← 0
07   c_1[s] ← r
08   BTreeSPLITChild(s, 1, r)
09   BTreeInsertNonFull(s, k)
10 else BTreeInsertNonFull(r, k)
```
Splitting the Root

- Splitting the root requires the creation of new nodes

- The tree grows at the top instead of the bottom
Insertion: Example

initial tree (t = 3)

B inserted

Q inserted
Insertion: Example (cont’)

L inserted

F inserted
Inserting Keys

- BtreeInsertNonFull tries to insert a key $k$ into a node $x$, which is **assumed to be nonfull** when the procedure is called.

- **BTreeInsert and the recursion in BTreeInsertNonFull** guarantee that this assumption is true!
Inserting Keys: Pseudo Code

BTreeInsertNonFull(x, k)

01 i ← n[x]
02 if leaf[x] then
03     while i ≥ 1 and k < key_i[x]
04         key_{i+1}[x] ← key_i[x]
05         i ← i - 1
06     key_{i+1}[x] ← k
07     n[x] ← n[x] + 1
08     DiskWrite(x)
09 else while i ≥ 1 and k < key_i[x]
10     do i ← i - 1
11     i ← i + 1
12     DiskRead c_i[x]
13     if n[c_i[x]] = 2t - 1 then
14         BTreeSplitChild(x, i, c_i[x])
15         if k > key_i[x] then
16             i ← i + 1
17         BTreeInsertNonFull(c_i[x], k)
Insertion: Running Time

- Disk I/O: $O(h)$, since only $O(1)$ disk accesses are performed during recursive calls of BTreeInsertNonFull

- CPU: $O(th) = O(t \log_t n)$

- At any given time there are $O(1)$ number of disk pages in main memory