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

Ladan Tahvildari, PEng, SMIEEE
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:

- http://www.stargroup.uwaterloo.ca/~ece250/acknowledgment/
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
  - 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 $key[x]$ refers to it
  - otherwise DiskRead($x$) reads the object from disk into main memory (DiskWrite($x$) – writes it back to disk)
Principles (2)

- A typical working pattern

01 ...
02 \( x \leftarrow \) a pointer to some object
03 \text{DiskRead}(x)
04 operations that access and/or modify \( x \)
05 \text{DiskWrite}(x) \ //\text{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
B-tree Definitions

- Node $x$ has fields
  - $n[x]$: the number of keys of that the node
  - $key_1[x] \leq \ldots \leq key_{n[x]}[x]$: the keys in ascending order
  - 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 $k_i \leq key_i[x] \leq k_{i+1}$
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:

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

\[
\begin{align*}
\sum_{i=1}^{h} 2t^{i-1} &= 2t^h - 1 \\
n \geq 1 + (t - 1) \sum_{i=1}^{h} 2t^{i-1} &= 2t^h - 1
\end{align*}
\]
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)

\[
\text{BTreeSearch}(x, k)
\]

1. \(i \leftarrow 1\)
2. \(\text{while } i \leq n[x] \text{ and } k > \text{key}_i[x]\)
3. \(i \leftarrow i+1\)
4. \(\text{if } i \leq n[x] \text{ and } k = \text{key}_i[x] \text{ then}\)
   - \(\text{return} (x, i)\)
5. \(\text{if leaf}[x] \text{ then}\)
   - \(\text{return} \text{NIL}\)
6. \(\text{else DiskRead}(c_i[x])\)
7. \(\text{return} \text{BTreeSearch}(c_i[x], k)\)
Creating an Empty Tree

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

\begin{verbatim}
BTreeCreate(T)
01  x ← AllocateNode();
02  leaf[x] ← TRUE;
03  n[x] ← 0;
04  DiskWrite(x);
05  root[T] ← x
\end{verbatim}
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
Splitting Nodes (cont’)

\begin{verbatim}
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+y
06 if not leaf[y] then
07    for j ← 1 to t
08        c_j[z] ← c_j+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)
\end{verbatim}

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

Video

Special case: root is full (BtreeInsert)

\[
\text{BTreeInsert}(T, k)
\]

01 \( r \leftarrow \text{root}[T] \)
02 \textbf{if} \( n[r] = 2t - 1 \) \textbf{then}
03 \( s \leftarrow \text{AllocateNode}() \)
04 \( \text{root}[T] \leftarrow s \)
05 \( \text{leaf}[s] \leftarrow \text{FALSE} \)
06 \( n[s] \leftarrow 0 \)
07 \( c_1[s] \leftarrow r \)
08 \text{BTreeSplitChild}(s, 1, r)
09 \text{BTreeInsertNonFull}(s, k)
10 \textbf{else} \text{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

\textbf{BTreeInsertNonFull}(x,k)

01 \( i \leftarrow n[x] \)
02 \textbf{if} leaf[x] \textbf{then}
03 \hspace{1em} \textbf{while} \( i \geq 1 \) and \( k < \text{key}_i[x] \)
04 \hspace{2em} \text{key}_{i+1}[x] \leftarrow \text{key}_i[x]
05 \hspace{2em} i \leftarrow i - 1
06 \hspace{1em} \text{key}_{i+1}[x] \leftarrow k
07 \hspace{1em} n[x] \leftarrow n[x] + 1
08 \hspace{1em} \text{DiskWrite}(x)
09 \textbf{else while} \( i \geq 1 \) and \( k < \text{key}_i[x] \)
10 \hspace{1em} \textbf{do} \( i \leftarrow i - 1 \)
11 \hspace{2em} i \leftarrow i + 1
12 \hspace{1em} \text{DiskRead} \ c_i[x]
13 \hspace{1em} \textbf{if} \ n[c_i[x]] = 2t - 1 \textbf{then}
14 \hspace{2em} \text{BTreeSplitChild}(x,i,c_i[x])
15 \hspace{1em} \textbf{if} \ k > \text{key}_i[x] \textbf{then}
16 \hspace{2em} i \leftarrow i + 1
17 \hspace{1em} \text{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