Memory Model

- For this course: Assume Uniform Access Time
  - All elements in an array accessible with same time cost
  - Reality is somewhat different
- Memory Hierarchy (in order of decreasing speed)
  - Registers
  - On (cpu) chip cache memory
  - Off chip cache memory
  - Main memory
  - Virtual memory (automatically managed use of disk)
  - Explicit disk I/O
- All but last managed by system
  - Need to be aware, but can do little to manipulate others directly
  - Promote locality?

Cost of Disk I/O

- Disk access 10,000 to 100,000 times slower than memory access
  - Do almost anything (almost!) in terms of computation to avoid an extra disk access
  - Performance penalty is huge
- B-Trees designed to be used with disk storage
  - Typically used with database application
  - Many different variations
  - Will present basic ideas here
- Want broad, not deep trees
  - Even log N disk accesses can be too many
  - When does the user notice a delay?

External Methods

- Disk use requires special consideration
  - Timing Considerations (already mentioned)
  - Writing pointers to disk?
  - What do values mean when read in at a different time/different machine?
- General Properties of B-Trees
  - All Leaves Have Same Depth
  - Degree of Node > 2
  - (maybe hundreds or thousands)
- Not a Binary Tree, but is a Search Tree
  - There are many implementations...
  - Will use examples with artificially small numbers to illustrate

Rules of B-Trees

- Rules
  1. Every node (except root) has at least MINIMUM entries
  2. The MAXIMUM number of node entries is 2*MINIMUM
  3. The entries of each B-tree are stored, sorted
  4. The number of sub-trees below a non-leaf node is one greater than the number of node entries
  5. For non leaves:
     - Entry at index k is greater than all entries in sub-tree k
     - Entry at index k is less than all entries in sub-tree k+1
  6. Every leaf in a B-tree has the same depth
### Example

**Example B Tree (MAX = 2)**

```
[6]
[2 4] [9]
[1] [3] [5] [7 8] [10]
```

### Search in B-Tree

- Every Child is Also the Root of a Smaller B-Tree
- Possible internal node implementation

```java
class BTNode {
    // ignoring ref on disk issue
    int myDataCount;
    int myChildCount;
    KeyType[] myKeys[MAX+1];
    BTNode[] myChild[MAX+2];
}
```

**Search:**

```java
boolean isInBTree(BTNode t, KeyType key);
1. Search through myKeys until myKeys[k] >= key
2. If t.myData[k] == key, return true
3. If isLeaf(t) return false
4. return isInBTree(t.myChild[k])
```

### Find Example

**Example Find in B-Tree (MAX = 2)**

```
[6 17]
[4] [12] [19 22]
[2 3] [5] [10] [16] [18] [20] [25]
```

### B-Tree Insertion

- Insertion Gets a Little Messy
  - Insertion may cause rule violation
  - “Loose” Insertion (leave extra space) (+1)
  - Fixing Excess Entries
- **Insert Fix**
  - Split
  - Move up middle
  - Height gained only at root
- **Look at some examples**
Insertion Fix

- (MAX = 4) Fixing Child with Excess Entry

\[
\begin{align*}
\text{BEFORE} & \quad 9 & 28 & 13 & 16 & 19 & 22 & 25 \\
\text{AFTER} & \quad 3 & 4 & 13 & 16 & 22 & 25 & 33 & 40
\end{align*}
\]

\[
\begin{align*}
\text{BEFORE} & \quad 2 & 3 & 4 & 5 & 7 & 8 & 11 & 12 & 14 & 15 & 17 & 18 & 20 & 21 & 23 & 24 & 26 & 27 & 31 & 32 & 34 & 35 & 50 & 51 \\
\text{AFTER} & \quad 9 & 19 & 28
\end{align*}
\]

Insertion Fix (MAX = 2) Another Fix

\[
\begin{align*}
\text{BEFORE} & \quad 6 & 17 & 4 & 12 & 18 & 19 & 22 \\
\text{AFTER} & \quad 6 & 17 & 4 & 12 & 19 & 22
\end{align*}
\]

B-Tree Removal

- Remove
  - Loose Remove
  - If rules violated: Fix
    - Borrow (rotation)
    - Join
- Examples left to the “reader”

B-Trees

- Many variations
  - Leaf node often different from internal node
  - Only leaf nodes carry all data (internal nodes: keys only)
  - Examples didn’t distinguish keys from data
  - Design to have nodes fit disk block
- The Big Picture
  - Details can be worked out
  - Can do a lot of computation to avoid a disk access