Lecture 12: B Trees
Outline

- B-tree
  - special case of multiway search trees
  - good performance if data needs to be accessed in large blocks, as opposed to random access
    - Example: storage on external disk
Reasons for using B-Trees

- Disk access is much slower than memory access
  - one disk access takes around 8ms
  - this is about the same as executing 200,000 CPU instructions
  - worth executing lots of instructions to avoid a single disk access

- Disk data is stored in blocks of size $B$, since access is slow
  - typical block size is between 1024 bytes and 8192 bytes
  - a whole block is read per one disk access

- When traversing a tree path, each node is likely to be in a different disk block
  - need to read $O(\log n)$ disk blocks
  - worth to reduce the tree height to avoid reading many disk blocks

- B-Tree has a high branching factor resulting in a smaller height
  - assuming a tree node fits into 1 disc block
B-Tree Definition

- B-tree of order $d$ is a multiway search tree such that
  1. Each internal node (except the root) has between $\lfloor d/2 \rfloor$ and $d$ children and, therefore, between $\lfloor d/2 \rfloor - 1$ to $d - 1$ entries
  2. all external nodes have the same depth

- B-tree of order 5
  - between 3 and 5 children

![B-Tree Diagram](image-url)
Why Root Node is Special

- Root is allowed fewer than \( \lceil \frac{d}{2} \rceil \) children (but at least 2 children) to allow for B-trees with a small number of entries.
- The only way to have a B-tree of order 5 storing 5 items.
B-Tree Height

- Height of a B-tree is
  \[ O \left( \log_{\left\lfloor \frac{d}{2} \right\rfloor} n \right) = O \left( \frac{\log n}{\log \left\lfloor \frac{d}{2} \right\rfloor} \right) \]

- Used property \( \frac{\log a}{\log b} = \log_b a \)

- The larger is \( d \), the shorter is the height
B-Tree Height

- B-tree of order 5

```
11    24
2   6   8   10
12   13  15
27    32   33
```

- B-tree of order 13

```
2   6   8   10   11   12   13   15   24   27   32   33
```

- $h = 3$
- $h = 2$
Operations in B-Tree

- Since B-tree is a multiway search tree, search performed exactly as in multiway search tree
- Insert and delete require only minor changes from that of (2,4) tree
  - perform splits as necessary when inserting
  - perform transfer and fusion as necessary when deleting
Insertion Example for B-tree with $d = 6$

- Internal nodes must have between 3 and 6 children
- Insert new entry in the node returned by search, like in (2,4)-trees
- Overflow occurs when a node becomes a 7-node

Insert new entry in the node returned by search, like in (2,4)-trees
Overflow occurs when a node becomes a 7-node
To handle overflow at node $v$, perform a split:

- Middle entry $m = \lfloor (d+1)/2 \rfloor$ in $v$ goes to the parent of $v$, in correct place.
- Entries $1$ through $\lfloor (d+1)/2 \rfloor - 1$ together with children $1$ through $\lfloor (d+1)/2 \rfloor$ go to new node $v'$, $v'$ becomes child to the left of $m$.
- Entries $\lfloor (d+1)/2 \rfloor + 1$ through $d$ together with children $\lfloor (d+1)/2 \rfloor + 1$ through $d+1$ go to new node $v''$, $v''$ becomes child to the right of $m$. 

**Insertion Example**

```
1 3 5
12 13 14 15 20 22
27 32 33

overflow!
```

```
11 24
11 14 24
12 13
15 20 22
27 32 33

after split
```
Delete

- Ensure that entry to be deleted is at node with leaf children
  - if not, swap entry with its inorder successor
- Delete the entry at node \( v \) and the appropriate leaf child
- Underflow occurs if number of children left is less than \( \lceil d/2 \rceil \)
- Handle underflow with either transfer and fusion (transfer is preferable)
  - **Transfer**, if an adjacent sibling \( w \) has at least \( \lceil d/2 \rceil + 1 \) entries
    - \( v \) gets an entry which is between \( v \) and \( w \) from its parent \( u \), sibling \( w \) gives the entry to parent \( u \) as a replacement, this is the entry of \( w \) which is closest to the entry moved from \( u \) to \( v \)
    - After transfer, no new underflows
  - **Fusion** if both adjacent siblings have less than \( \lceil d/2 \rceil + 1 \) entries
    - merge node \( u \) with an adjacent sibling \( w \) into new entry \( v' \)
    - \( v' \) gets an entry from the parent. This is the entry between the old nodes \( v \) and \( w \)
    - After fusion, underflow at a parent node may occur
Deletion Example, \( d = 6 \)

```
underflow!
```

```
delete 11
```

```
transfer
```

```
derelation
```

```
1 3 5
```

```
12 24
```

```
5 24
```

```
1 3
```

```
12 14
```

```
27 32 33
```

```
27 32 33
```

```
27 32 33
```

```
14
```

```
27 32 33
```

```
27 32 33
```

```
1 3
```

```
12 14
```

```
27 32 33
```

```
1 3
```
Deletion Example, $d = 6$

11 24

2 4

12 14

25 26

2

12 14

25 26

11 24

2 11 12 14

25 26

underflow!
Complexity Analysis for Main Memory

- Assume time spent at each node is $O(d)$
  - can store entries at each node as an ordered dictionary using search table
- Height of B-tree with $n$ entries is
  $$O\left(\frac{\log n}{\log \left\lceil d/2 \right\rceil}\right)$$
- Thus search, insert, and delete take
- If the tree does fit into memory, better use (2,4)-tree, where search, insert, and delete have complexity $O(\log n)$
  $$O\left(d \frac{\log n}{\log \left\lceil d/2 \right\rceil}\right) = O\left(\log n \cdot \frac{d}{\log \left\lceil d/2 \right\rceil}\right)$$
Complexity Analysis for Disk

- $f$ is time for one CPU operation, and $s$ be the time for 1 disc access
  - $f$ is much smaller than $s$, approximately $s = 200,000 f$
- When performing search, insert, and delete we spent $d$ time at each node, and perform $\log n \over \log \lceil d/2 \rceil$ disk accesses
- Time to perform search, insert, and delete is
  \[ Time(d) = f \cdot \frac{\log n}{\log \lceil d/2 \rceil} + s \cdot \frac{\log n}{\log \lceil d/2 \rceil} \]
  - fast main memory operations
  - slow disk operations
- Suppose $f = 1$, $s = 200,000$. What $d$ to minimizes the $Time(d)$?
- Taking derivative of $Time(d)$ and setting it to 0, get $d = 4000$
- Should make $d = 4000$
  - provided a node storing 4000 entries fits into disc block size
Comments on Storage

- Choose $d$ so that $d$ children references and the $d - 1$ entries stored at a node can fit into a single disk block.

- Since each node has at least $\lceil d/2 \rceil$ children, each disk block used to support B-tree is at least half full.
  - Not much disk space is wasted by being empty.
Number of Entries in a Multiway Tree

The number of entries in a multiway tree is the number of external nodes – 1

Proof: Put one stone on each leaf and pass the stones from the leaves up the tree. Each internal node keeps 1 stone per entry, and passes the rest to its parent. Since # entries = # children – 1, each internal node will pass only 1 stone to its parent. This process stops at the root, and the root will pass 1 stone outside the tree. At the end, each entry has 1 stone, and 1 stone is outside the tree.
Number of Entries in a Multiway Tree

2 6 8 10

12 13 15

27 32 33

11 24
Number of Entries in a Multiway Tree
Reasons for using B-Trees

- If use B-tree of order 200, node size is usually small enough to transfer each node in one disc read operation.
- B-tree of order 200 and height 3 can store up to $200^3 - 1$ entries (approximately 8 million) and any entry can be accessed with 2 disc reads.
  - Assuming the root is stored in main memory, which makes sense due to its frequent access.

```
Level 0: 200^0 = 1 internal nodes
Level 1: 200^1 internal nodes
Level 2: 200^2 internal nodes
Level 3: 200^3 external nodes
```
Tree Summary (done with Trees)

- Binary Trees
  - Heaps
    - Balanced, but allows only to prioritise (not order) the keys
  - Binary search tree
    - Not necessarily balanced
    - If keys are inserted/deleted at random, has performance can be close to optimal, $O(\log n)$.
    - However in the worst case, performance is $O(n)$
  - AVL-tree
    - Guaranteed to be balanced with $O(\log n)$ performance, but more complicated to maintain than BST

- Multi-way trees
  - (2,4) tree is balanced. Guaranteed $O(\log n)$ time performance
  - B-Trees is balanced, useful for dictionary located in external (disk) memory