COMP 2140 - Data Structures

Shahin Kamali

Topic 10 - B-Trees
University of Manitoba

Partially based on notes by S. Durocher.
Overview

- Introduction to 2-3 trees
- b-trees as an extension of 2-3 trees
- Dictionary Operations on 2-3 trees and b-trees
A **ternary tree** is a tree in which each node has at most 3 children.

A 2-3 Tree is a ternary tree like a BST with additional structural properties:

- Every node either contains **one KVP** and **two children**, or **two KVPs** and **three children**.
- All the leaves are at the same level (A leaf is a node with empty children.)

![2-3 Tree Diagram](image-url)
Search in a 2-3 tree

- Searching through a 1-node is just like in a BST.
- For a 2-node, we must examine both keys and follow the appropriate path.

`search(21)`
Search in a 2-3 tree

- Searching through a 1-node is just like in a BST.
- For a 2-node, we must examine both keys and follow the appropriate path.

`search(21)`
Searching through a 1-node is just like in a BST.

For a 2-node, we must examine both keys and follow the appropriate path.

```
search(21)
```

```
25 43
```

```
18
```

```
21 24
```

```
28
```

```
31 36
```

```
39 42
```

```
48
```

```
56 62
```

```
12
```

```
33
```

```
42
```

```
51
```
Insertion in a 2-3 tree

Inserting a new KVP to a 2-3 tree

- First, we search to find the leaf where the new key belongs.
- If the leaf has only 1 KVP, just add the new one to make a 2-node.
- Otherwise, order the three keys as \( a < b < c \).
  Split the leaf into two 1-nodes, containing \( a \) and \( c \), and (recursively) insert \( b \) into the parent along with the new link.

Example: \textit{insert}(19)
Insertion in a 2-3 tree

- Inserting a new KVP to a 2-3 tree
  - First, we search to find the leaf where the new key belongs.
  - If the leaf has only 1 KVP, just add the new one to make a 2-node.
  - Otherwise, order the three keys as $a < b < c$.
    Split the leaf into two 1-nodes, containing $a$ and $c$, and (recursively) insert $b$ into the parent along with the new link.

Example: $insert(19)$

```
      25 43
    /   \
18    31 36
/ 
12 21 24 28 33 39 42 48 51
```

```
      25 43
    /   \
18    31 36
/ 
12 21 24 28 33 39 42 48 56 62
```
Insertion in a 2-3 tree

- Inserting a new KVP to a 2-3 tree
  - First, we search to find the leaf where the new key belongs.
  - If the leaf has only 1 KVP, just add the new one to make a 2-node.
  - Otherwise, order the three keys as \( a < b < c \).
    Split the leaf into two 1-nodes, containing \( a \) and \( c \),
    and (recursively) insert \( b \) into the parent along with the new link.

Example: \( \text{insert}(19) \)
Inserting a new KVP to a 2-3 tree

First, we search to find the leaf where the new key belongs.
If the leaf has only 1 KVP, just add the new one to make a 2-node.
Otherwise, order the three keys as $a < b < c$.
Split the leaf into two 1-nodes, containing $a$ and $c$, and (recursively) insert $b$ into the parent along with the new link.

Example: $\text{insert}(19)$
2-3 Trees

Insertion in a 2-3 tree

Inserting a new KVP to a 2-3 tree
- First, we search to find the leaf where the new key belongs.
- If the leaf has only 1 KVP, just add the new one to make a 2-node.
- Otherwise, order the three keys as $a < b < c$.
  Split the leaf into two 1-nodes, containing $a$ and $c$, and (recursively) insert $b$ into the parent along with the new link.

Example: $\text{insert}(41)$
Insertion in a 2-3 tree

Inserting a new KVP to a 2-3 tree

- First, we search to find the leaf where the new key belongs.
- If the leaf has only 1 KVP, just add the new one to make a 2-node.
- Otherwise, order the three keys as \( a < b < c \).
  Split the leaf into two 1-nodes, containing \( a \) and \( c \),
  and (recursively) insert \( b \) into the parent along with the new link.

Example: \textit{insert}(41)
Insertion in a 2-3 tree

- Inserting a new KVP to a 2-3 tree
  - First, we search to find the leaf where the new key belongs.
  - If the leaf has only 1 KVP, just add the new one to make a 2-node.
  - Otherwise, order the three keys as $a < b < c$.
    Split the leaf into two 1-nodes, containing $a$ and $c$, and (recursively) insert $b$ into the parent along with the new link.

Example: \textit{insert}(41)
Insertion in a 2-3 tree

Inserting a new KVP to a 2-3 tree

- First, we search to find the leaf where the new key belongs.
- If the leaf has only 1 KVP, just add the new one to make a 2-node.
- Otherwise, order the three keys as \( a < b < c \).
  - Split the leaf into two 1-nodes, containing \( a \) and \( c \),
  - and (recursively) insert \( b \) into the parent along with the new link.

Example: \( \text{insert}(41) \)
2-3 Trees

Insertion in a 2-3 tree

- Inserting a new KVP to a 2-3 tree
  - First, we search to find the leaf where the new key belongs.
  - If the leaf has only 1 KVP, just add the new one to make a 2-node.
  - Otherwise, order the three keys as \( a < b < c \).
    Split the leaf into two 1-nodes, containing \( a \) and \( c \),
    and (recursively) insert \( b \) into the parent along with the new link.

Example: \( \text{insert}(41) \)
Deletion from a 2-3 Tree

As with BSTs and AVL trees, we first swap the KVP with its successor → this way we always delete from a leaf.

Say we’re deleting KVP \( x \) from a node \( V \):

- If \( V \) is a 2-node, just delete \( x \).
- Else If \( V \) has a 2-node immediate sibling \( U \), perform a transfer:
  Put the “intermediate” KVP in the parent between \( V \) and \( U \) into \( V \), and replace it with the adjacent KVP from \( U \).
- Otherwise, we merge \( V \) and a 1-node sibling \( U \):
  Remove \( V \) and (recursively) delete the “intermediate” KVP from the parent, adding it to \( U \).
Example: \textit{delete}(43)
Example: delete(43)
2-3 Tree Deletion

Example: delete(43)
Example: delete(19)
2-3 Trees

2-3 Tree Deletion

Example: \textit{delete}(19)
2-3 Trees

2-3 Tree Deletion

Example: delete(19)
2-3 Trees

2-3 Tree Deletion

Example: \textit{delete}(42)
2-3 Trees

2-3 Tree Deletion

Example: $\text{delete}(42)$
Example: \textit{delete}(42)
2-3 Tree Deletion

Example: \textit{delete}(42)
2-3 Trees

2-3 Tree Deletion

Example: delete(42)
A **B-tree of minsize** $d$ is a search tree satisfying:

- Each node contains at most $2d$ KVPs.
  - Non-root nodes contain at least $d$ KVPs (root can have 1 or more).
- All the leaves are at the same level.

Some people call this a B-tree of order $(2d + 1)$, or a $(d + 1, 2d + 1)$-tree.

- The 2-3 Tree is a specific type of B-tree with $d = 1$.
- Here is a tree with $d = 2$: 

```
   24
  /|
 / |
10 18
 / | 
5 7 12 17
 / | 
19 21 22
 / | 
27 32
 / |
35 42 50
 / | 
38 39 46 49
 / 
60 70 80
```
**B-Tree Operations**

- *search, insert, delete* work just like for 2-3 trees.
  - As before, *insert might* result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).
  - For *delete*, if there is an overflow, we check if any direct sibling has an extra key; if it does not, we merge by creating a node containing the underflowed node (with \(d - 1\) keys), the key at parent (1 key), and direct sibling (\(d\) keys). The new key will have size \(2d\).

Insert (90)
B-Trees

B-Tree Operations

- search, insert, delete work just like for 2-3 trees.
  - As before, insert might result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).
  - For delete, if there is an overflow, we check if any direct sibling has an extra key; if it does not, we merge by creating a node containing the underflowed node (with $d - 1$ keys), the key at parent (1 key), and direct sibling ($d$ keys). The new key will have size $2d$.

insert (90)
B-Trees

B-Tree Operations

- *search, insert, delete* work just like for 2-3 trees.
  - As before, *insert* might result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).
  - For *delete*, if there is an overflow, we check if any direct sibling has an extra key; if it does not, we merge by creating a node containing the underflowed node (with $d - 1$ keys), the key at parent (1 key), and direct sibling ($d$ keys). The new key will have size $2d$.

```
insert (91)
```

```
5 7 12 17 19 21 22 27 32 38 39 46 49 60 70 80 90
```
B-Tree Operations

- \textit{search, insert, delete} work just like for 2-3 trees.
  - As before, \textit{insert might} result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).
  - For \textit{delete}, if there is an overflow, we check if any direct sibling has an extra key; if it does not, we merge by creating a node containing the underflowed node (with $d - 1$ keys), the key at parent (1 key), and direct sibling ($d$ keys). The new key will have size $2d$.

![B-Tree Diagram]

**insert (91)**
B-Trees

B-Tree Operations

- *search, insert, delete* work just like for 2-3 trees.
  - As before, *insert* might result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).
  - For *delete*, if there is an overflow, we check if any direct sibling has an extra key; if it does not, we merge by creating a node containing the underflowed node (with \(d - 1\) keys), the key at parent (1 key), and direct sibling (\(d\) keys). The new key will have size \(2d\).

```plaintext
insert (91)
```

```
                  24
               /   \   \
             10     35 42 50 80
            / \   / \        /
           5  7 12 17 19 21 22 27 32 38 39 46 49 60 70 90 91
```
B-Trees

B-Tree Operations

- *search, insert, delete* work just like for 2-3 trees.
  - As before, *insert might* result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).
  - For *delete*, if there is an overflow, we check if any direct sibling has an extra key; if it does not, we merge by creating a node containing the underflowed node (with $d - 1$ keys), the key at parent (1 key), and direct sibling ($d$ keys). The new key will have size $2d$.

```
delete (5)
```

![B-Tree diagram]

```
24
/  \
10 18
/  \
5 7 12 17
/  \
19 21 22
/  \
27 32
/  \
35 42 50
/  \\n38 39 46 49 60 70
/  \
90 91
```
B-Trees

B-Tree Operations

- **search**, **insert**, **delete** work just like for 2-3 trees.
  - As before, **insert might** result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).
  - For **delete**, if there is an overflow, we check if any direct sibling has an extra key; if it does not, we merge by creating a node containing the underflowed node (with \( d - 1 \) keys), the key at parent (1 key), and direct sibling (\( d \) keys). The new key will have size \( 2d \).
**B-Trees**

**B-Tree Operations**

- *search, insert, delete* work just like for 2-3 trees.
  - As before, *insert might* result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).
  - For *delete*, if there is an overflow, we check if any direct sibling has an extra key; if it does not, we merge by creating a node containing the underflowed node (with \( d - 1 \) keys), the key at parent (1 key), and direct sibling (\( d \) keys). The new key will have size \( 2d \).

**delete (5)**

```
7 12 17 19 21 22 27 32 38 39 46 49 60 70 90 91
7 10 18 35 42 50 80
10 18
24
```
B-Trees

B-Tree Operations

- **search, insert, delete** work just like for 2-3 trees.
  - As before, *insert might* result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).
  - For *delete*, if there is an overflow, we check if any direct sibling has an extra key; if it does not, we merge by creating a node containing the underflowed node (with $d - 1$ keys), the key at parent (1 key), and direct sibling ($d$ keys). The new key will have size $2d$.

delete (5)
B-Trees

B-Tree Operations

- *search, insert, delete* work just like for 2-3 trees.
  - As before, *insert might* result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).
  - For *delete*, if there is an overflow, we check if any direct sibling has an extra key; if it does not, we merge by creating a node containing the underflowed node (with \(d - 1\) keys), the key at parent (1 key), and direct sibling (\(d\) keys). The new key will have size \(2d\).

![B-Tree Diagram](image)
### Height of a B-tree

What is the least number of KVPs in a height-\(h\) B-tree?

<table>
<thead>
<tr>
<th>Level</th>
<th>#Nodes is (\ge)</th>
<th>Node size is (\ge)</th>
<th>KVPs is (\ge)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>(d)</td>
<td>(2d)</td>
</tr>
<tr>
<td>2</td>
<td>(2(d+1))</td>
<td>(d)</td>
<td>(2d(d+1))</td>
</tr>
<tr>
<td>3</td>
<td>(2(d+1)^2)</td>
<td>(d)</td>
<td>(2d(d+1)^2)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>(h)</td>
<td>(2(d+1)^{h-1})</td>
<td>(d)</td>
<td>(2d(d+1)^{h-1})</td>
</tr>
</tbody>
</table>

Total: \(n \ge 1 + \sum_{i=0}^{h-1} 2d(d+1)^i = 2(d+1)^h - 1\)

\[
\log(n+1) \ge 1 + h \log(d+1) \rightarrow h \le \frac{\log(n+1) - 1}{\log(d+1)} = O\left(\frac{\log n}{\log d}\right)
\]
Assume each node stores its KVPs and child-pointers in a dictionary that supports $O(\log d)$ search, insert, and delete.

Then search, insert, and delete work just like for 2-3 trees, and each require $\Theta(\text{height})$ node operations.

Total cost is $O\left(\frac{\log n}{\log d} \cdot (\log d)\right) = O(\log n)$. 
Tree-based data structures have poor memory locality: If an operation accesses $m$ nodes, then it must access $m$ spaced-out memory locations.

**Observation**: Accessing a single location in external memory (e.g., hard disk) automatically loads a whole block (or “page”).

In an AVL tree or 2-3 tree, $\Theta(\log n)$ pages are loaded in the worst case for a single insert/delete/search operation.

If $d$ is small enough so a $2d$-node fits into a single page, then a B-tree of minsize $d$ only loads $\Theta((\log n)/(\log d))$ pages.

This can result in a huge savings: memory access is often the largest time cost in a computation.
Max size $2d + 1$: Permitting one additional KVP in each node allows *insert* and *delete* to avoid *backtracking* via *pre-emptive splitting* and *pre-emptive merging*.

**Red-black trees**: Identical to a B-tree with minsize 1 and maxsize 3, but each 2-node or 3-node is represented by 2 or 3 binary nodes, and each node holds a “color” value of red or black.

**B⁺-trees**: All KVPs are stored at the leaves (interior nodes just have keys), and the leaves are linked sequentially.