For efficient multi-way tree implementation, we need small $h$ and $d$. What is the best trade-off?
Performance depends on $h$ and $d$

For efficient multi-way tree implementation, we need small $h$ and $d$. What is the best trade-off?

(2,4) Trees

- Achieve $h = \Theta(\log(n))$ and $2 \leq d \leq 4$.
- **Size Property**: every node has at most four children.
- **Depth Property**: all the external nodes have the same depth.
- Size and Depth Properties $\Rightarrow h = \Theta(\log(n))$. 
Insertion in a (2,4) Tree

Insert key $k$

- Search for that key $k$. 

Depth Property preserved! Might violate the Size Property! Overflow! A generic way to handle the overflow needed!
Insertion in a (2,4) Tree

Insert key $k$

- Search for that key $k$.
- If no such $k$, the search terminates at an external node $z$. 

Depth Property preserved! Might violate the Size Property! overflow! A generic way to handle the overflow needed!
Insertion in a (2,4) Tree

Insert key $k$

- Search for that key $k$.
- If no such $k$, the search terminates at an external node $z$.
- Let $v$ be $z$’s parent. The new item is inserted into $v$. 
Insertion in a (2,4) Tree

Insert key \( k \)

- Search for that key \( k \).
- If no such \( k \), the search terminates at an external node \( z \).
- Let \( v \) be \( z \)'s parent. The new item is inserted into \( v \).
- Depth Property preserved! Might violate the Size Property! overflow!
Insertion in a (2,4) Tree

Insert key $k$

- Search for that key $k$.
- If no such $k$, the search terminates at an external node $z$.
- Let $v$ be $z$’s parent. The new item is inserted into $v$.
- Depth Property preserved! Might violate the Size Property! overflow!
- A generic way to handle the overflow needed!
When overflow, $\nu$ must be a 5-node. Let $\nu_1, \cdots, \nu_5$ be its children. Let $k_1 \leq k_2 \leq k_3 \leq k_4$ be the keys stored in $\nu$.
Restoration at Overflow!

When overflow, \( v \) must be a 5-node. Let \( v_1, \cdots, v_5 \) be its children. Let \( k_1 \leq k_2 \leq k_3 \leq k_4 \) be the keys stored in \( v \).

**Split operation on** \( v \)

- \( v \rightarrow v', v'' \): \( v' \), 3-node with \( v_1, v_2, v_3 \) and \( k_1, k_2 \); \( v'' \), 2-node with \( v_4, v_5 \) and \( k_4 \).
When overflow, \( v \) must be a 5-node. Let \( v_1, \ldots, v_5 \) be its children. Let \( k_1 \leq k_2 \leq k_3 \leq k_4 \) be the keys stored in \( v \).

**Split operation on** \( v \)

- \( v \rightarrow v', v'' \): \( v' \), 3-node with \( v_1, v_2, v_3 \) and \( k_1, k_2 \); \( v'' \), 2-node with \( v_4, v_5 \) and \( k_4 \).
- Let \( u \) be \( v' \)’s parent if exists. Otherwise, create a parent (root) \( u \).
Restoration at Overflow!

When overflow, \( v \) must be a 5-node. Let \( v_1, \ldots, v_5 \) be its children. Let \( k_1 \leq k_2 \leq k_3 \leq k_4 \) be the keys stored in \( v \).

**Split operation on \( v \)**

- \( v \rightarrow v', v'' \): \( v' \), 3-node with \( v_1, v_2, v_3 \) and \( k_1, k_2 \); \( v'' \), 2-node with \( v_4, v_5 \) and \( k_4 \).
- Let \( u \) be \( v \)'s parent if exists. Otherwise, create a parent (root) \( u \).
- Insert \( k_3 \) into \( u \), and attach \( v', v'' \) to \( u \) accordingly.
Restoration at Overflow!

When overflow, $v$ must be a 5-node. Let $v_1, \cdots, v_5$ be its children. Let $k_1 \leq k_2 \leq k_3 \leq k_4$ be the keys stored in $v$.

Split operation on $v$

- $v \rightarrow v', v'': v'$, 3-node with $v_1, v_2, v_3$ and $k_1, k_2$; $v''$, 2-node with $v_4, v_5$ and $k_4$.
- Let $u$ be $v$'s parent if exists. Otherwise, create a parent (root) $u$.
- Insert $k_3$ into $u$, and attach $v', v''$ to $u$ accordingly.

This might cause $u$ to overflow, repeat the same procedure again until no overflow.
(2,4) Tree insertion

Insertion one by one: 4, 6, 12, 15, 3, 5, 10, 8.
(2,4) Tree insertion

Insertion one by one: 4, 6, 12, 15, 3, 5, 10, 8.
(2,4) Tree insertion: insert 17
(2,4) Tree insertion: insert 17
(2,4) Tree insertion: insert 17
(2,4) Tree insertion: insert 17
Correctness and Complexity

Correctness

- Each split restores the size property of current nodes and preserve the depth property.
Correctness and Complexity

Correctness

- Each split restores the size property of current nodes and preserve the depth property.
- Might cause new violations, however, only among its ancestors. Only $O(h) = O(\log(n))$ such splits.
Correctness and Complexity

Correctness

- Each split restores the size property of current nodes and preserve the depth property.
- Might cause new violations, however, only among its ancestors. Only $O(h) = O(\log(n))$ such splits.

Complexity

- Each split takes $O(1)$. In total, $O(\log(n))$ such splits.
- Total time is $O(\log(n))$. 
Removal

Remove key $k$

- Search for that key $k$. 

Claim: can always reduce to case (1). Suppose the key is the $i$th item $k_i$ at a node $z$. Find the right-most internal node $v$ in the subtree rooted at the $i$th child of $z$. Claim: $v$ is case (1). Why?

Swap $k_i$ and the last item of $v$. Reduce to case (1).
Removal

Remove key $k$

- Search for that key $k$.
- If no such $k$, nothing to remove.
Remove key \( k \)

- Search for that key \( k \).
- If no such \( k \), nothing to remove.
- Otherwise, arrive at an internal node (two cases): (1) with all external nodes (2) otherwise.

Claim: can always reduce to case (1). Suppose the key is the \( i \)th item \( k_i \) at a node \( z \).

Find the right-most internal node \( v \) in the subtree rooted at the \( i \)th child of \( z \). Claim: \( v \) is case (1). Why?

Swap \( k_i \) and the last item of \( v \). Reduce to case (1).
Removal

Remove key $k$

- Search for that key $k$.
- If no such $k$, nothing to remove.
- Otherwise, arrive at an internal node (two cases): (1) with all external nodes (2) otherwise.
- Claim: can always reduce to case (1). Suppose the key is the $i$th item $k_i$ at a node $z$.
  - Find the right-most internal node $v$ in the subtree rooted at the $i$th child of $z$. Claim: $v$ is case (1). Why?
Removal

Remove key $k$

- Search for that key $k$.
- If no such $k$, nothing to remove.
- Otherwise, arrive at an internal node (two cases): (1) with all external nodes (2) otherwise.
- Claim: can always reduce to case (1). Suppose the key is the $i$th item $k_i$ at a node $z$.
  - Find the right-most internal node $v$ in the subtree rooted at the $i$th child of $z$. Claim: $v$ is case (1). Why?
  - Swap $k_i$ and the last item of $v$. Reduce to case (1).
Removal: Cont’d

Remove key $k$: Case 1

- $k$ is stored at a node $v$ with only external children.
Removal: Cont’d

Remove key $k$: Case 1

- $k$ is stored at a node $v$ with only external children.
- Remove it. The depth property is preserved.
Removal: Cont’d

Remove key \( k \): Case 1

- \( k \) is stored at a node \( v \) with only external children.
- Remove it. The depth property is preserved.
- However, the size property might be violated (i.e., under-flow)
Remove key $k$: Case 1

- $k$ is stored at a node $v$ with only external children.
- Remove it. The depth property is preserved.
- However, the size property might be violated (i.e., under-flow)
- A generic way to handle the under-flow needed.
Find the immediate siblings of $v$

- Note $v$ should be 2-node before removal.
Restoration at Underflow!

Find the immediate siblings of $v$

- Note $v$ should be 2-node before removal.
- If there is an immediate sibling $w$ of $v$ (3-node or 4-node), then perform a \texttt{transfer} operation.
Restoration at Underflow!

Find the immediate siblings of $v$

- Note $v$ should be 2-node before removal.
- If there is an immediate sibling $w$ of $v$ (3-node or 4-node), then perform a transfer operation.
- Otherwise, perform a fusion operation with an immediate sibling $w$ of $v$. (2-node in this case).
Transfer operation

$v$: node to remove keys, $w$: 3-node or 4-node and $v$’s immediate sibling, $u$: $w$, $v$’s parent and the key $k$ that separates $v, w$.

- assume $w$ is after $v$, similarly for the other case.

Let $k_w$ be the first key in $w$ and $T_w$ the first subtree of $w$.

Move $k_w$ to $u$, replacing the position of $k$.

Move $k$ to $v$ as the last key.

Move $T_w$ to be the last subtree of $v$.

Correctness & Implementation

Transfer preserves the depth-property and the multi-way search tree property.

Restore the size-property of $v$. Preserve the size-property of the rest nodes.

Implementation: $O(1)$. 
Transfer operation

\( v \): node to remove keys, \( w \): 3-node or 4-node and \( v \)'s immediate sibling, \( u \): \( w \), \( v \)'s parent and the key \( k \) that separates \( v, w \).

- assume \( w \) is after \( v \), similarly for the other case.
- Let \( k_w \) be the first key in \( w \) and \( T_w \) the first subtree of \( w \).
Transfer operation

\( \nu \): node to remove keys, \( \omega \): 3-node or 4-node and \( \nu \)'s immediate sibling, \( u \): \( \omega \), \( \nu \)'s parent and the key \( k \) that separates \( \nu, \omega \).

- assume \( \omega \) is after \( \nu \), similarly for the other case.
- Let \( k_\omega \) be the first key in \( \omega \) and \( T_\omega \) the first subtree of \( \omega \).
- Move \( k_\omega \) to \( u \), replacing the position of \( k \). Move \( k \) to \( \nu \) as the last key.

Correctness & Implementation

- Transfer preserves the depth-property and the multi-way search tree property.
- Restore the size-property of \( \nu \). Preserve the size-property of the rest nodes.
- Implementation: \( O(1) \).
Transfer operation

\( v \): node to remove keys, \( w \): 3-node or 4-node and \( v \)'s immediate sibling, \( u \): \( w \), \( v \)'s parent and the key \( k \) that separates \( v, w \).

- assume \( w \) is after \( v \), similarly for the other case.
- Let \( k_w \) be the first key in \( w \) and \( T_w \) the first subtree of \( w \).
- Move \( k_w \) to \( u \), replacing the position of \( k \). Move \( k \) to \( v \) as the last key.
- Move \( T_w \) to be the last subtree of \( v \).

Correctness & Implementation

- Transfer preserves the depth-property and the multi-way search tree property.
- Restore the size-property of \( v \). Preserve the size-property of the rest nodes.
- Implementation: \( O(1) \).
Transfer operation

\( v \): node to remove keys, \( w \): 3-node or 4-node and \( v \)’s immediate sibling, \( u \): \( w \), \( v \)’s parent and the key \( k \) that separates \( v, w \).

- assume \( w \) is after \( v \), similarly for the other case.
- Let \( k_w \) be the first key in \( w \) and \( T_w \) the first subtree of \( w \).
- Move \( k_w \) to \( u \), replacing the position of \( k \). Move \( k \) to \( v \) as the last key.
- Move \( T_w \) to be the last subtree of \( v \).

Correctness & Implementation

- **Transfer** preserves the depth-property and the multi-way search tree property.
Transfer operation

\( \nu: \) node to remove keys, \( \omega: \) 3-node or 4-node and \( \nu \)'s immediate sibling, \( \omega: \) \( \nu \)'s parent and the key \( k \) that separates \( \nu, \omega \).

- Assume \( \omega \) is after \( \nu \), similarly for the other case.
- Let \( k_{\omega} \) be the first key in \( \omega \) and \( T_{\omega} \) the first subtree of \( \omega \).
- Move \( k_{\omega} \) to \( \omega \), replacing the position of \( k \). Move \( k \) to \( \nu \) as the last key.
- Move \( T_{\omega} \) to be the last subtree of \( \nu \).

Correctness & Implementation

- **Transfer** preserves the depth-property and the multi-way search tree property.
- Restore the size-property of \( \nu \). Preserve the size-property of the rest nodes.
- Implementation: \( O(1) \).
(2,4) Tree insertion: Remove 4
(2,4) Tree insertion: Remove 4
(2,4) Tree insertion: Remove 4
(2,4) Tree insertion: Remove 4
(2,4) Tree insertion: Remove 12 (reduction to case 1)
(2,4) Tree insertion: Remove 12 (reduction to case 1)