CS 240 – Data Structures and Data Management

Module 11: External Memory

Mark Petrick, Éric Schost
Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Spring 2024

version 2024−07−29 23:13

Outline

11 External Memory
- Motivation
- Stream-based algorithms
- External Dictionaries
  - *a*-*b*-trees
  - 2-4-trees and Red-Black Trees
  - B-trees
- External Hashing

Outline

11 External Memory
- Motivation
- Stream-based algorithms
- External Dictionaries
  - *a*-*b*-trees
  - 2-4-trees and Red-Black Trees
  - B-trees
- External Hashing

Different levels of memory

Recall the RAM model of a computer: Any access to a memory location takes the same (constant) time.

This is not at all realistic!

Different levels of memory

Recall the RAM model of a computer: Any access to a memory location takes the same (constant) time.

This is not at all realistic!

A typical current computer architecture includes

- registers (very fast, very small)
- cache L1, L2 (still fast, less small)
- main memory
- disk or cloud (slow, very large)

Different levels of memory

Recall the RAM model of a computer: Any access to a memory location takes the same (constant) time.

This is not at all realistic!

A typical current computer architecture includes

- registers (very fast, very small)
- cache L1, L2 (still fast, less small)
- main memory
- disk or cloud (slow, very large)

General question: how to adapt our algorithms to take the memory hierarchy into account, avoiding transfers as much as possible?

Different levels of memory

Recall the RAM model of a computer: Any access to a memory location takes the same (constant) time.

This is not at all realistic!

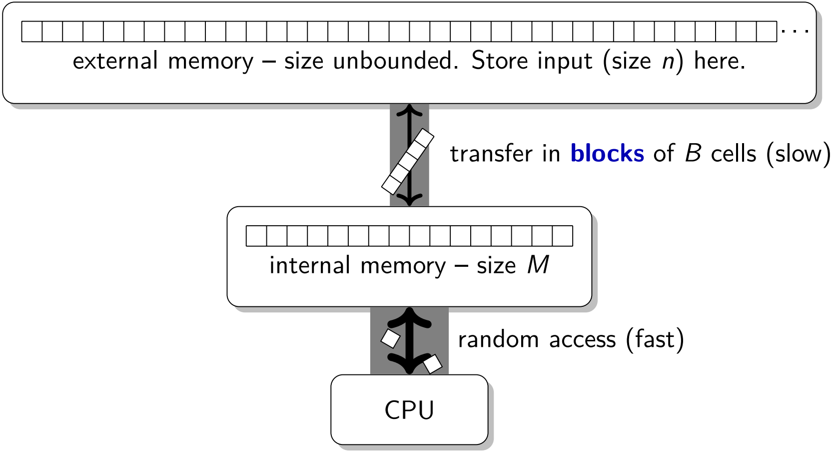
A typical current computer architecture includes

- registers (very fast, very small)
- cache L1, L2 (still fast, less small)
- main memory
- disk or cloud (slow, very large)

General question: how to adapt our algorithms to take the memory hierarchy into account, avoiding transfers as much as possible?

Define a new computer model that models one such 'gap' across which we must transfer.

The External-Memory Model (EMM)



Assumption: During a *transfer*, we automatically load a whole **block** (or "page"). This is quite realistic.

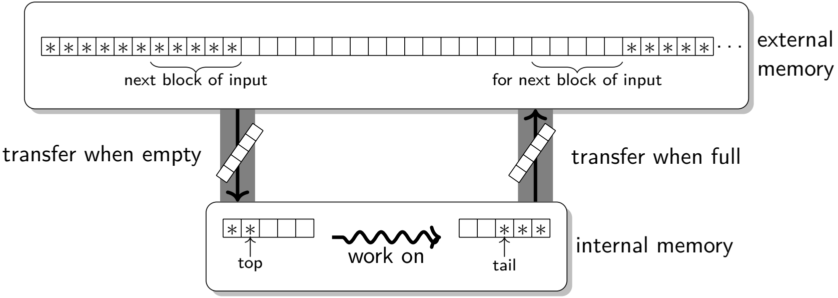
New objective: revisit all algorithms/data structures with the objective of minimizing **block transfers** ("probes", "disk transfers", "page loads")

Outline

11 External Memory
- Motivation
- Stream-based algorithms
- External Dictionaries
  - *a*-*b*-trees
  - 2-4-trees and Red-Black Trees
  - B-trees
- External Hashing

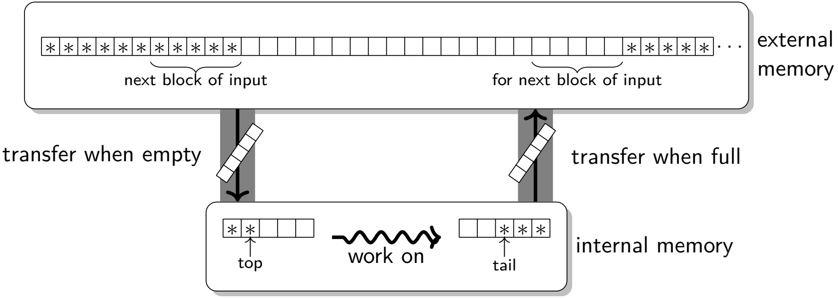
Streams and external memory

Stream-based algorithms (with $O(1)$ resets) use $\Theta(\frac{n}{B})$ block transfers.



Streams and external memory

Stream-based algorithms (with $O(1)$ resets) use $\Theta(\frac{n}{B})$ block transfers.



So can do the following with $\Theta(\frac{n}{B})$ block transfers:

- Text compression: Huffman, run-length encoding, Lempel-Ziv-Welch
- Pattern matching: Karp-Rabin, Knuth-Morris-Pratt, Boyer-Moore
  (This assumes internal memory has $O(|P|)$ space.)
- Sorting: *merge* can be implemented with streams
  $\rightsquigarrow$ *merge-sort* uses $O(\frac{n}{B}\log n)$ block transfers (can be improved)

Outline

11 External Memory
- Motivation
- Stream-based algorithms
- External Dictionaries
  - *a*-*b*-trees
  - 2-4-trees and Red-Black Trees
  - B-trees
- External Hashing

Dictionaries in external memory

Recall: Dictionaries store n KVPs and support *search*, *insert* and *delete*.

- **Recall:** AVL-trees were optimal in time and space in RAM model
- $\Theta(\log n)$ run-time $\Rightarrow$ $O(\log n)$ block transfers per operation
- But: Inserts happen at varying locations of the tree.
  $\rightsquigarrow$ nearby nodes are unlikely to be on the same block
  $\rightsquigarrow$ typically $\Theta(\log n)$ block transfers per operation

Dictionaries in external memory

Recall: Dictionaries store n KVPs and support *search*, *insert* and *delete*.

- **Recall:** AVL-trees were optimal in time and space in RAM model
- $\Theta(\log n)$ run-time $\Rightarrow$ $O(\log n)$ block transfers per operation
- But: Inserts happen at varying locations of the tree.
  $\rightsquigarrow$ nearby nodes are unlikely to be on the same block
  $\rightsquigarrow$ typically $\Theta(\log n)$ block transfers per operation
- We would like to have *fewer* block transfers.
  - Goal: $O(\log_B n)$ block transfers.
  - Does this really make a difference?
  - Consider 'typical' values: $n \approx 2^{50}$, $B \approx 2^{15}$.
    What is $\log n$ vs. $\log_B n$?

Dictionaries in external memory

Recall: Dictionaries store n KVPs and support *search*, *insert* and *delete*.

- **Recall:** AVL-trees were optimal in time and space in RAM model
- $\Theta(\log n)$ run-time $\Rightarrow$ $O(\log n)$ block transfers per operation
- But: Inserts happen at varying locations of the tree.
  $\rightsquigarrow$ nearby nodes are unlikely to be on the same block
  $\rightsquigarrow$ typically $\Theta(\log n)$ block transfers per operation
- We would like to have *fewer* block transfers.
  - Goal: $O(\log_B n)$ block transfers.
  - Does this really make a difference?
  - Consider 'typical' values: $n \approx 2^{50}$, $B \approx 2^{15}$.
    What is $\log n$ vs. $\log_B n$?

Better solution: design a tree-structure that *guarantees* that many nodes on search-paths are within one block.

Idealized structure

Idea: Store complete subtrees with $\log b$ levels in one block of memory.
($b \in \Theta(B)$ is maximal so that these fit into one block.)

- Each block/subtree then covers height $\log b$

$\Rightarrow$ Search-path hits $\frac{\log n}{\log b}$ blocks $\Rightarrow \log_b n$ block-transfers

- Since $b \in \Theta(B)$, we have $\log_b n \in \Theta(\log_B n)$ (why?)

Idealized structure

Idea: Store complete subtrees with $\log b$ levels in one block of memory.
($b \in \Theta(B)$ is maximal so that these fit into one block.)

- Each block/subtree then covers height $\log b$

$\Rightarrow$ Search-path hits $\frac{\log n}{\log b}$ blocks $\Rightarrow$ $\log_b n$ block-transfers

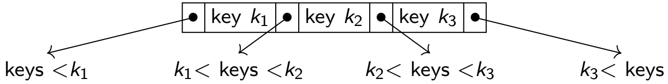
- Since $b \in \Theta(B)$, we have $\log_b n \in \Theta(\log_B n)$ (why?)

Idea: View the entire content of a block as one node.

Towards a-b-trees

Define *multiway-tree*: A node can store multiple keys.

Definition: A d**-node** stores d keys, has $d+1$ subtrees, and stored keys are between the keys in the subtrees.

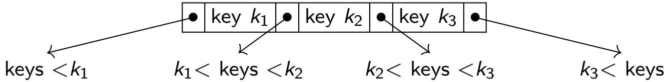


We *always* have one more subtree than keys (but subtrees may be empty).

Towards *a*-*b*-trees

Define *multiway-tree*: A node can store multiple keys.

Definition: A ***d*-node** stores d keys, has $d+1$ subtrees, and stored keys are between the keys in the subtrees.



We *always* have one more subtree than keys (but subtrees may be empty).

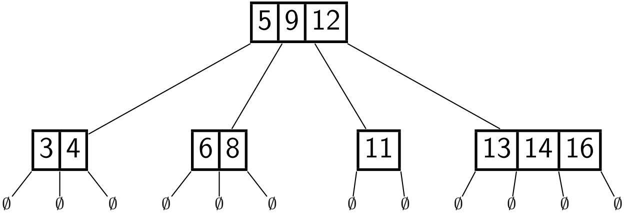
- To allow *insert*/*delete*, we permit a varying numbers of keys in nodes (within limits)
- We also rigidly restrict where empty subtrees may be.
- This gives much smaller height than for AVL-trees
  $\Rightarrow$ fewer block transfers

a-b-trees

Definition: An a-b-**tree** (for some $b \geq 3$ and $2 \leq a \leq \lceil \frac{b}{2} \rceil$) satisfies

1. Every non-root is a d-node for some $a-1 \leq d \leq b-1$.
   - Between a and b subtrees, between $a-1$ and $b-1$ keys.
2. The root is a d-node for $1 \leq d \leq b-1$.
   - Between 2 and b subtrees, between 1 and $b-1$ keys.
3. All empty subtrees are at the same level.

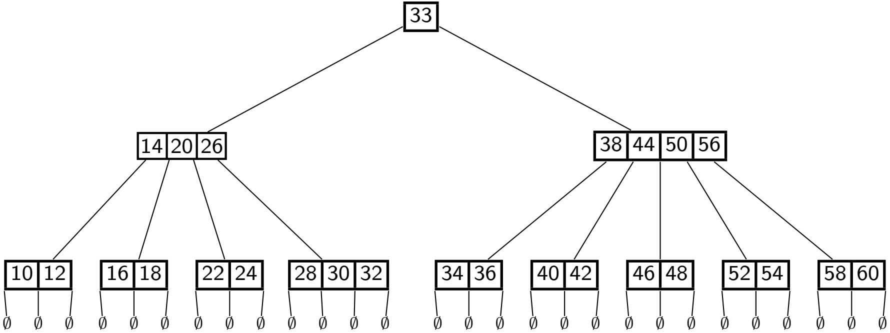
Example: A 2-4-tree of height 1.



For 2-4-trees, every node has between 1 and 3 keys.

a-b-tree Example

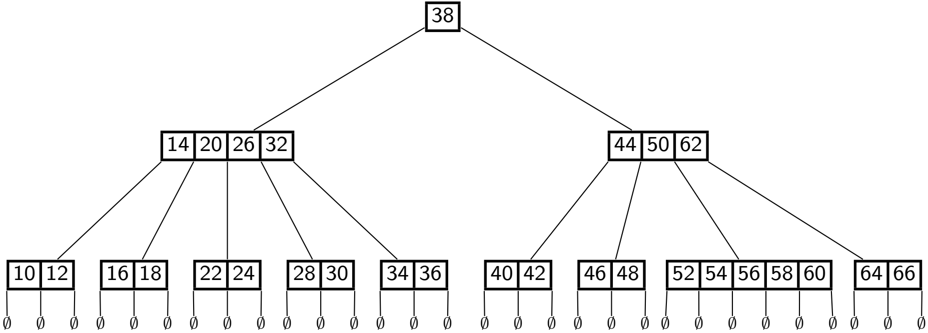
Example: A 3-5-tree of height 2.



Typically we will specify the **order** b and then set $a = \lceil \frac{b}{2} \rceil$.

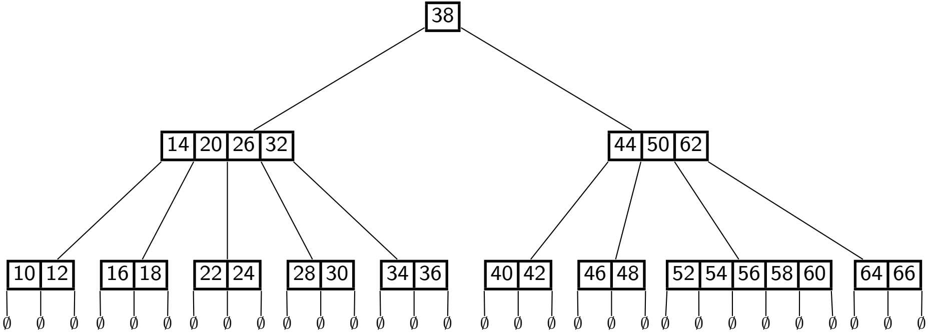
a-*b*-tree Example

Example: A 3-6-tree of height 2.



a-b-tree Example

Example: A 3-6-tree of height 2.



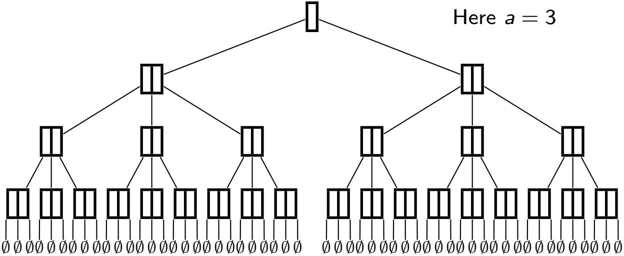
Note: With small height we can store *many* keys.
A 3-6-tree of height 2 can store up to $(1 + 6 + 36) \cdot 5 = 215$ keys.

a-b-tree Height

Theorem: An a-b-tree with n keys has $O(\log_a(n))$ height.

Proof: How many keys *must* an a-b-tree of height h have?

Level	**Nodes**
1	≥ 2
2	$\geq 2a$
3	$\geq 2a^2$
$\vdots$	$\vdots$
h	$\geq 2a^{h-1}$

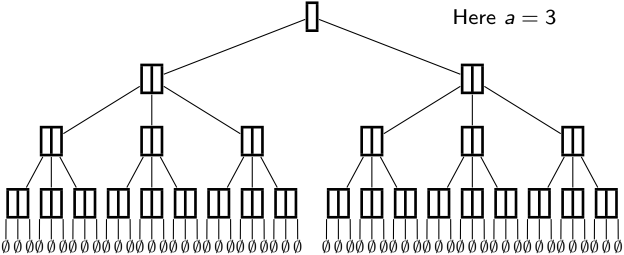


a-b-tree Height

Theorem: An a-b-tree with n keys has $O(\log_a(n))$ height.

Proof: How many keys *must* an a-b-tree of height h have?

Level	Nodes
1	≥ 2
2	$\geq 2a$
3	$\geq 2a^2$
$\vdots$	$\vdots$
h	$\geq 2a^{h-1}$



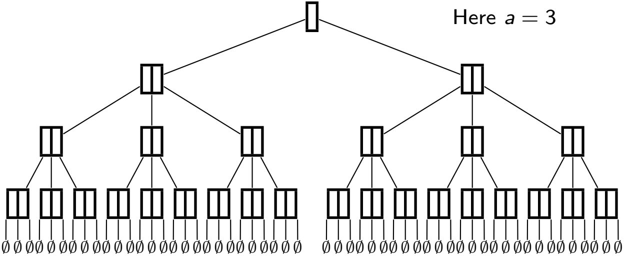
$$\#\text{ non-root nodes} \quad \geq \quad \sum_{i=1}^{h} 2a^{i-1} = 2\sum_{j=0}^{h-1} a^j = 2\frac{a^h - 1}{a - 1}$$

a-b-tree Height

Theorem: An a-b-tree with n keys has $O(\log_a(n))$ height.

Proof: How many keys *must* an a-b-tree of height h have?

Level	Nodes
1	≥ 2
2	$\geq 2a$
3	$\geq 2a^2$
$\vdots$	$\vdots$
h	$\geq 2a^{h-1}$



Here $a = 3$

$$\#\text{ non-root nodes} \geq \sum_{i=1}^{h} 2a^{i-1} = 2\sum_{j=0}^{h-1} a^j = 2\frac{a^h - 1}{a-1}$$

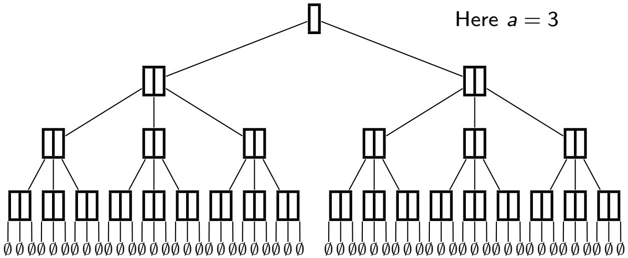
$$n = \#\text{ KVPs} \geq \underbrace{1}_{\text{root}} + \underbrace{(a-1)}_{\geq a-1 \text{ KVPs at non-root}} 2\frac{a^h - 1}{a-1} = 2a^h - 1$$

a-b-tree Height

Theorem: An a-b-tree with n keys has $O(\log_a(n))$ height.

Proof: How many keys *must* an a-b-tree of height h have?

Level	Nodes
1	≥ 2
2	$\geq 2a$
3	$\geq 2a^2$
$\vdots$	$\vdots$
h	$\geq 2a^{h-1}$



$$\#\text{ non-root nodes} \geq \sum_{i=1}^{h} 2a^{i-1} = 2\sum_{j=0}^{h-1} a^j = 2\frac{a^h - 1}{a-1}$$

$$n = \#\text{ KVPs} \geq \underbrace{1}_{\text{root}} + \underbrace{(a-1)}_{\geq a-1 \text{ KVPs at non-root}} 2\frac{a^h-1}{a-1} = 2a^h - 1$$

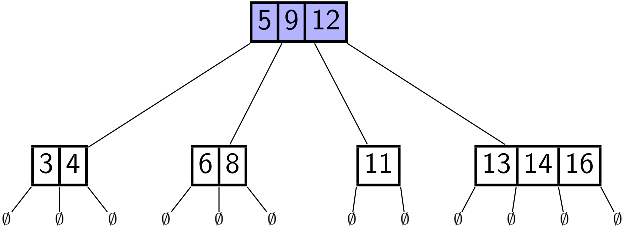
Therefore $h \leq \log_a\left(\frac{n+1}{2}\right)$.

a-*b*-tree Operations

Search is similar to BST:

- Compare search-key to keys at node
- If not found, continue in appropriate subtree until empty

Example: *search*(15)

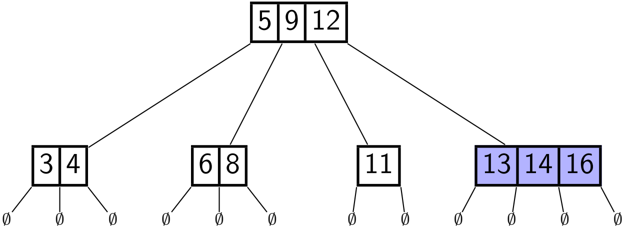


a-*b*-tree Operations

Search is similar to BST:

- Compare search-key to keys at node
- If not found, continue in appropriate subtree until empty

Example: *search*(15)

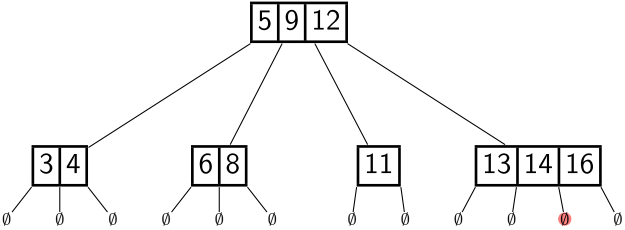


a-*b*-tree Operations

Search is similar to BST:

- Compare search-key to keys at node
- If not found, continue in appropriate subtree until empty

Example: *search*(15) *not found*



a-*b*-tree *search*

abTree::search(k)

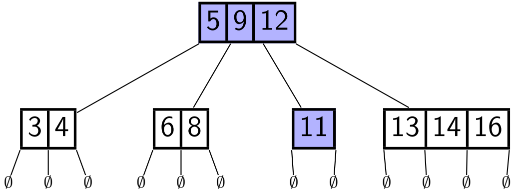
$z \leftarrow$ *root*, $p \leftarrow$ `NULL`   `// p: parent of z`
while z is not `NULL`
    let $\langle T_0, k_1, \ldots, k_d, T_d \rangle$ be key-subtree list at z
    if $k \geq k_1$
        $i \leftarrow$ maximal index such that $k_i \leq k$
        if $k_i = k$ **then return** KVP at k_i
        else $p \leftarrow z$, $z \leftarrow$ root of T_i
    else $p \leftarrow z$, $z \leftarrow$ root of T_0
return "not found, would be in p"

- # visited nodes: $O(\log_a n)$ (one per level)
- Note: Finding i is not constant time (depending on b)

a-b tree *insert*

- Do *abTree::search* and add key and empty subtree at leaf.

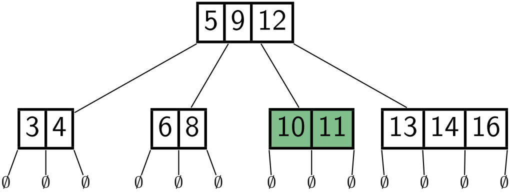
Example (2-4-tree): *insert*(10)



a-b tree *insert*

- Do *abTree::search* and add key and empty subtree at leaf.
- If the leaf had room then we are done.

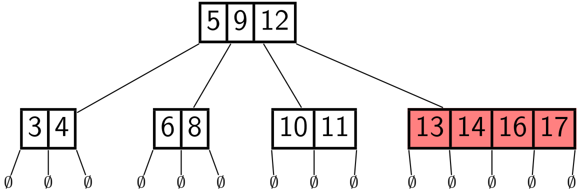
Example (2-4-tree): *insert*(10)



a-b tree *insert*

- Do *abTree::search* and add key and empty subtree at leaf.
- If the leaf had room then we are done.
- Else **overflow**: More keys/subtrees than permitted.
- Resolve overflow by **node splitting**.

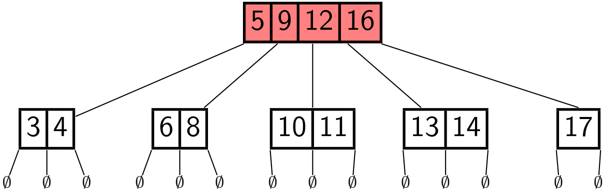
Example (2-4-tree): *insert*(17)



a-b tree *insert*

- Do *abTree::search* and add key and empty subtree at leaf.
- If the leaf had room then we are done.
- Else **overflow**: More keys/subtrees than permitted.
- Resolve overflow by **node splitting**.

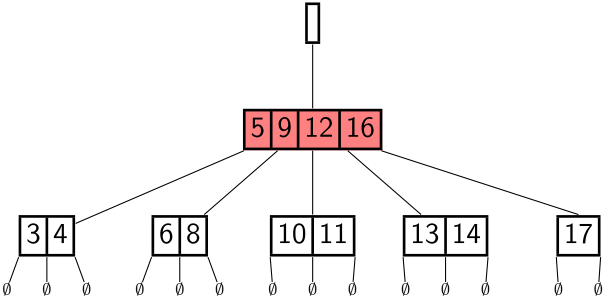
Example (2-4-tree): *insert*(17)



a-b tree *insert*

- Do *abTree::search* and add key and empty subtree at leaf.
- If the leaf had room then we are done.
- Else **overflow**: More keys/subtrees than permitted.
- Resolve overflow by **node splitting**.

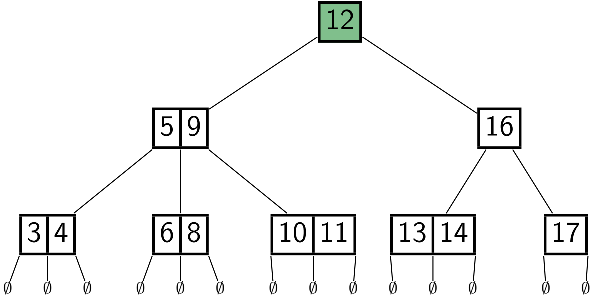
Example (2-4-tree): *insert*(17)



a-*b* tree *insert*

- Do *abTree::search* and add key and empty subtree at leaf.
- If the leaf had room then we are done.
- Else **overflow**: More keys/subtrees than permitted.
- Resolve overflow by **node splitting**.

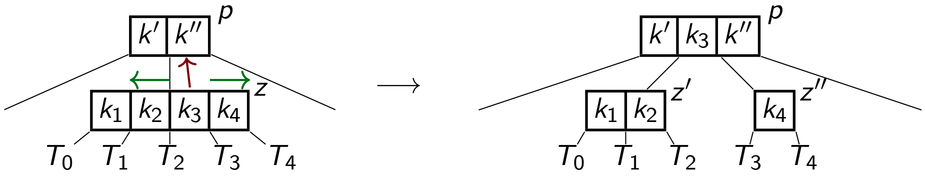
Example (2-4-tree): *insert*(17)



a-*b*-tree *insert*

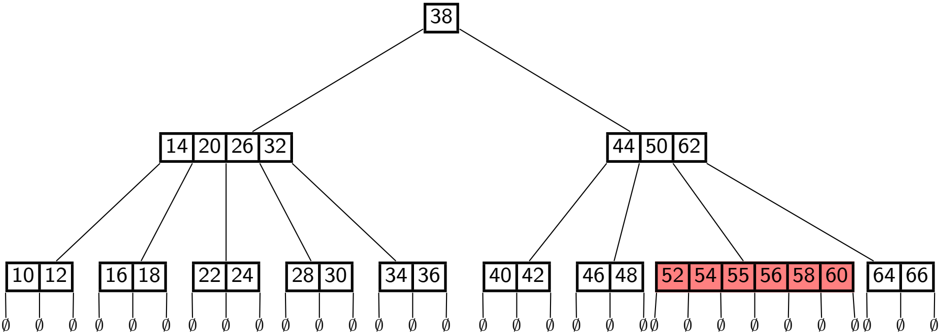
abTree::insert(k)

```
z ← abTree::search(k)  // z: leaf where k should be
Add k and an empty subtree in key-subtree-list of z
while z has b keys (overflow ⇝ node split)
    Let ⟨T_0, k_1, ..., k_b, T_b⟩ be key-subtree list at v
    if (z has no parent) create a parent of z without KVPs
    move upper median k_m of keys to parent p of z
    z' ← new node with ⟨T_0, k_1, ..., k_{m-1}, T_{m-1}⟩
    z'' ← new node with ⟨T_m, k_{m+1}, ..., k_b, T_b⟩
    Replace ⟨z⟩ by ⟨z', k_m, z''⟩ in key-subtree-list of p
    z ← p
```



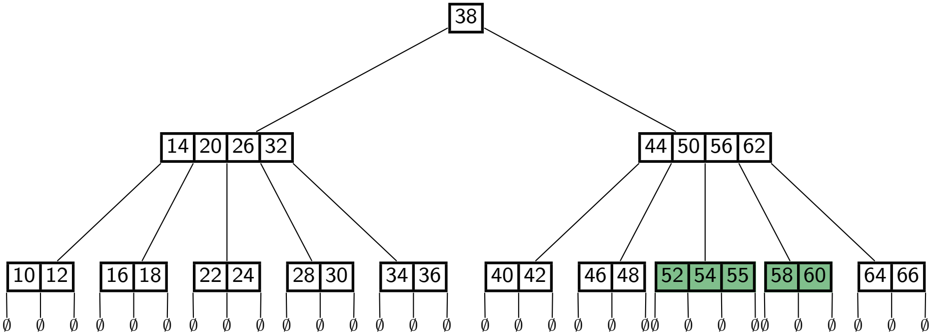
a-b-tree *insert*

Example: *insert*(55) in a 3-6-tree:



a-b-tree *insert*

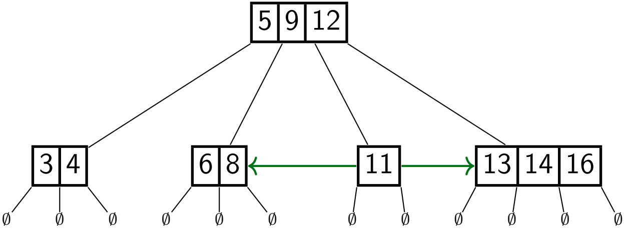
Example: *insert*(55) in a 3-6-tree:



- Node split $\Rightarrow$ new nodes have $\geq \lfloor (b-1)/2 \rfloor = \lceil b/2 \rceil - 1$ keys
- Since we know $a \leq \lceil b/2 \rceil$, this is $\geq a-1$ keys as required.

Towards 2-4-tree Deletion

- For deletion, we symmetrically will have to handle **underflow** (too few keys/subtrees)
- Crucial ingredient for this: **immediate sibling**

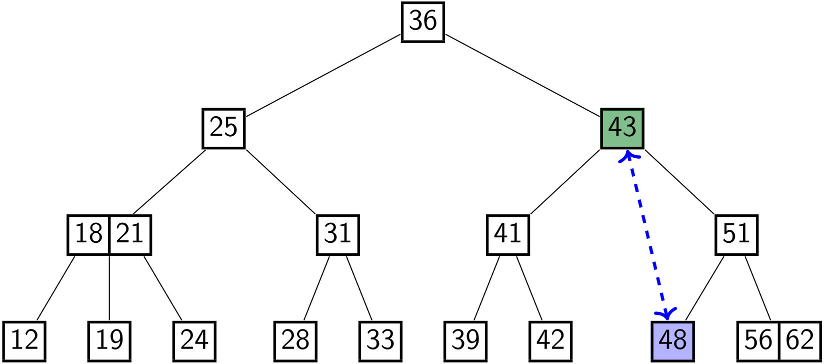


- **Observe:** Any node except the root has an immediate sibling.

2-4-tree Deletion

Example: *delete*(43)

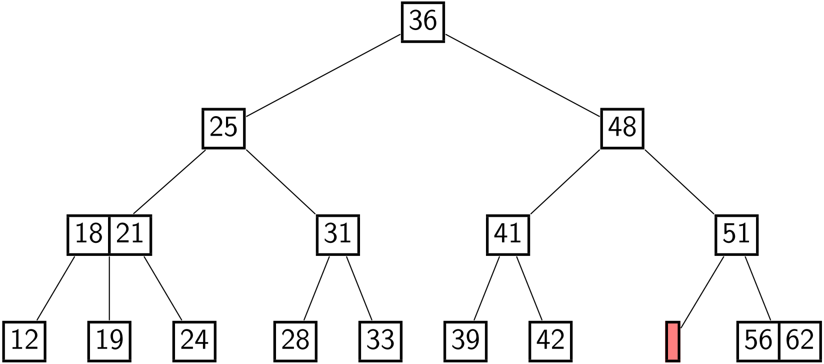
- *abTree::search*, then trade with successor if KVP is not at a leaf.



2-4-tree Deletion

Example: *delete*(43)

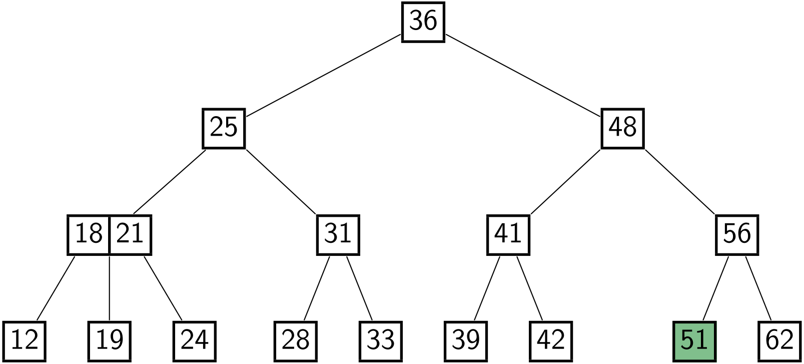
- *abTree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:



2-4-tree Deletion

Example: *delete*(43)

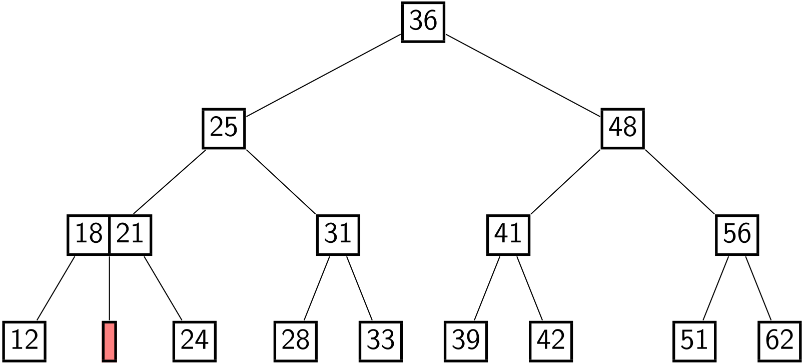
- *abTree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:
  - If immediate sibling has extras, **rotate/transfer**



2-4-tree Deletion

Example: *delete*(19)

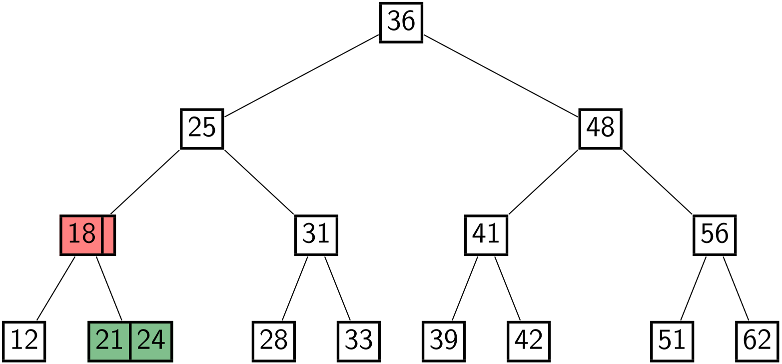
- *abTree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:
  - If immediate sibling has extras, **rotate/transfer**



2-4-tree Deletion

Example: *delete*(19)

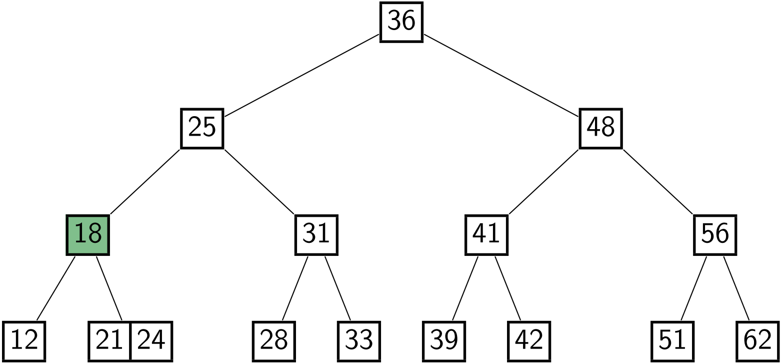
- *abTree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:
  - If immediate sibling has extras, **rotate/transfer**
  - Else **node merge** (this affects the parent!)



2-4-tree Deletion

Example: *delete*(19)

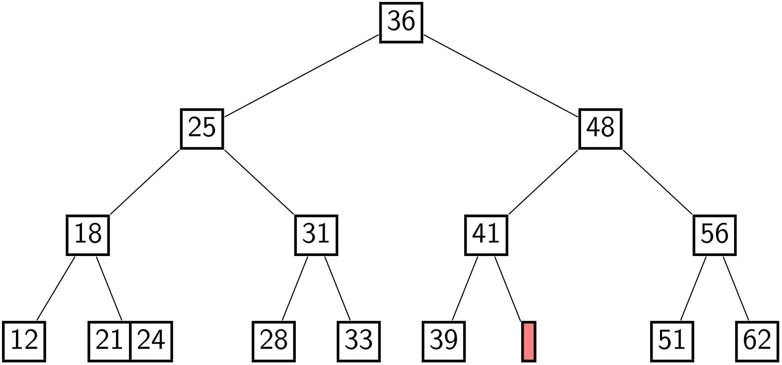
- *abTree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:
  - If immediate sibling has extras, **rotate/transfer**
  - Else **node merge** (this affects the parent!)



2-4-tree Deletion

Example: *delete*(42)

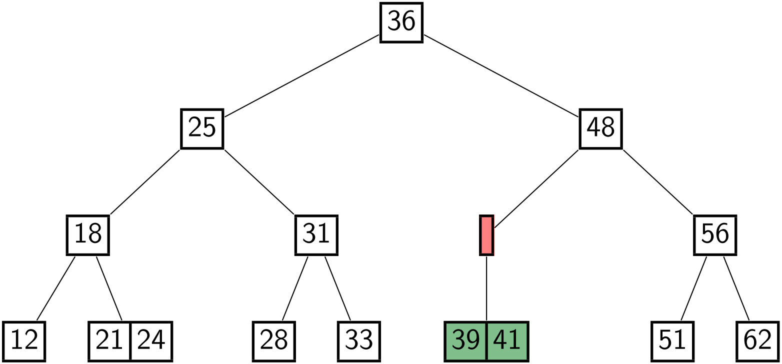
- *abTree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:
  - If immediate sibling has extras, **rotate/transfer**
  - Else **node merge** (this affects the parent!)



2-4-tree Deletion

Example: *delete*(42)

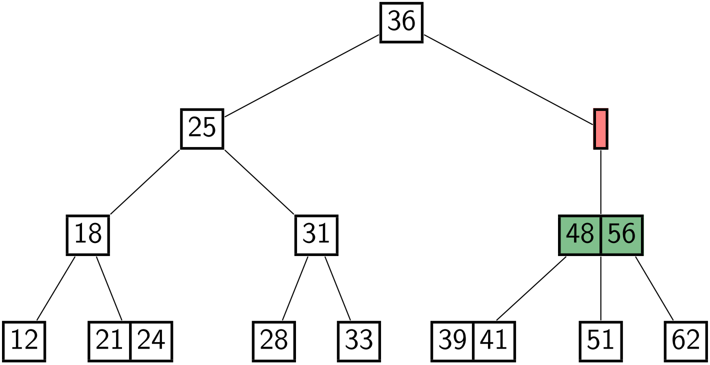
- *abTree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:
  - If immediate sibling has extras, **rotate/transfer**
  - Else **node merge** (this affects the parent!)



2-4-tree Deletion

Example: *delete*(42)

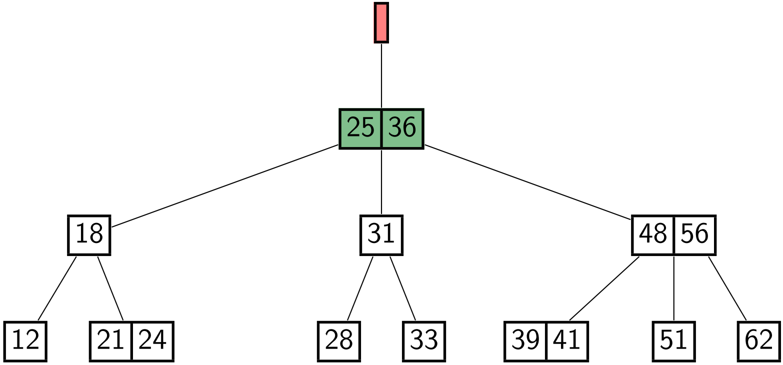
- *abTree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:
  - If immediate sibling has extras, **rotate/transfer**
  - Else **node merge** (this affects the parent!)



2-4-tree Deletion

Example: *delete*(42)

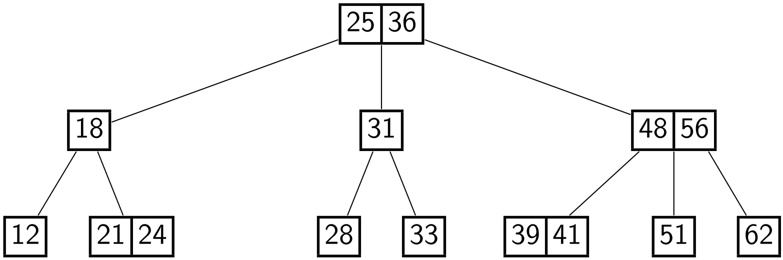
- *abTree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:
  - If immediate sibling has extras, **rotate/transfer**
  - Else **node merge** (this affects the parent!)



2-4-tree Deletion

Example: *delete*(42)

- *abTree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:
  - If immediate sibling has extras, **rotate/transfer**
  - Else **node merge** (this affects the parent!)



Deletion from a 2-4-tree

```
24Tree::delete(k)
v ← 24Tree::search(k) // node containing k
if v is not leaf
    swap k with its successor k′ and v with leaf containing k′
delete k and one empty subtree in v
while v has 0 keys (underflow)
    if parent p of v is NULL, delete v and break
    if v has immediate sibling u with 2 or more keys (transfer/rotate)
        transfer the key of u that is nearest to v to p
        transfer the key of p between u and v to v
        transfer the subtree of u that is nearest to v to v
        break
    else (merge & repeat)
        u ← immediate sibling of v
        transfer the key of p between u and v to u
        transfer the subtree of v to u
        delete node v and set v ← p
```

a-b-tree Summary

- An a-b tree has height $O(\log_a n)$
- If $a \approx b/2$, then this height-bound is tight.
  - Level i contains at most b^i nodes
  - Each node contains at most $b - 1$ KVPs
  - So $n \leq b^{h+1} - 1$ and $h \in \Omega(\log_b n)$.

a-*b*-tree Summary

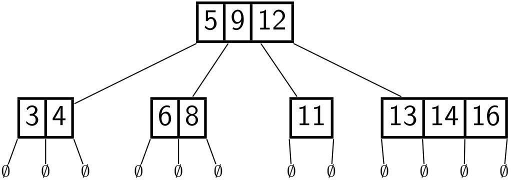
- An a-b tree has height $O(\log_a n)$
- If $a \approx b/2$, then this height-bound is tight.
  - Level i contains at most b^i nodes
  - Each node contains at most $b-1$ KVPs
  - So $n \leq b^{h+1}-1$ and $h \in \Omega(\log_b n)$.
- *search* and *insert* visit $O(\log_a n)$ nodes.
- *delete* can also be implemented with $O(\log_a n)$ node-visits.
  But usually use *lazy deletion*—space is cheap in external memory.

a-b-tree Summary

- An a-b tree has height $O(\log_a n)$
- If $a \approx b/2$, then this height-bound is tight.
  - Level i contains at most b^i nodes
  - Each node contains at most $b-1$ KVPs
  - So $n \leq b^{h+1} - 1$ and $h \in \Omega(\log_b n)$.
- *search* and *insert* visit $O(\log_a n)$ nodes.
- *delete* can also be implemented with $O(\log_a n)$ node-visits.
  But usually use *lazy deletion*—space is cheap in external memory.
- How do we choose the order b? (Recall: a is usually $\lceil \frac{b}{2} \rceil$.)
  - Option 1: b small, e.g. $b = 4$
    $\rightsquigarrow$ a new balanced BST, competetive with AVL-trees.
  - Option 2: b big (but one node still fits into one block of memory)
    $\rightsquigarrow$ a realization of ADT Dictionary for external memory

2-4-trees

Consider the special case of $b = 4$ (hence $a = 2$):

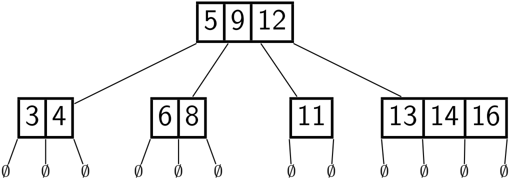


- We analyze here the runtime in the RAM-model (include cost of operations in internal memory)
- Height is $O(\log n)$, operations visit $O(\log n)$ nodes.
- Each node stores $O(1)$ keys and subtrees, so $O(1)$ time spent at node.

$\Rightarrow$ All operations take $O(\log n)$ **worst-case time**.

2-4-trees

Consider the special case of $b = 4$ (hence $a = 2$):



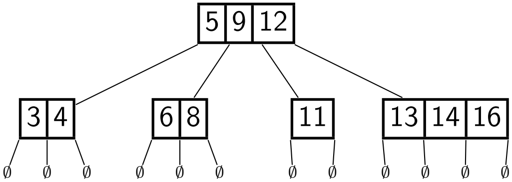
- We analyze here the runtime in the RAM-model (include cost of operations in internal memory)
- Height is $O(\log n)$, operations visit $O(\log n)$ nodes.
- Each node stores $O(1)$ keys and subtrees, so $O(1)$ time spent at node.

$\Rightarrow$ All operations take $O(\log n)$ **worst-case time**.

This is the same as AVL-trees in theory.
But we can make them even better in practice.

Towards red-black-trees

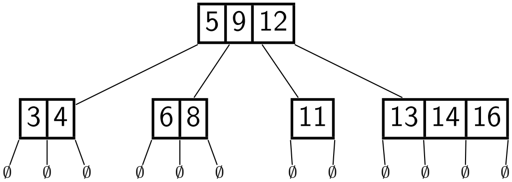
Problems with 2-4-trees:



- Recall: We have three kinds of nodes (1-node, 2-node, 3-node) so up to 7 items (keys and subtree-references) at a node.
- *insert* can change the number of keys and subtrees at a node.
- How should we store key-subtree list?

Towards red-black-trees

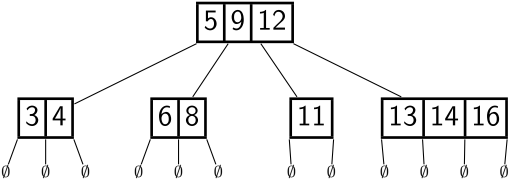
Problems with 2-4-trees:



- Recall: We have three kinds of nodes (1-node, 2-node, 3-node) so up to 7 items (keys and subtree-references) at a node.
- *insert* can change the number of keys and subtrees at a node.
- How should we store key-subtree list?
  - Array? Then we must use length 7. *This wastes space.*

Towards red-black-trees

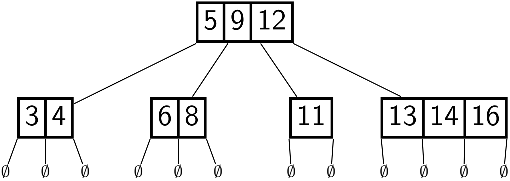
Problems with 2-4-trees:



- Recall: We have three kinds of nodes (1-node, 2-node, 3-node) so up to 7 items (keys and subtree-references) at a node.
- *insert* can change the number of keys and subtrees at a node.
- How should we store key-subtree list?
  - Array? Then we must use length 7. *This wastes space.*
  - Linked list? We have overhead for list-nodes. *This wastes space.*

Towards red-black-trees

Problems with 2-4-trees:

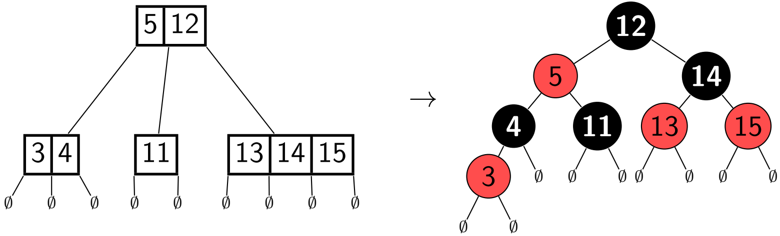


- Recall: We have three kinds of nodes (1-node, 2-node, 3-node) so up to 7 items (keys and subtree-references) at a node.
- *insert* can change the number of keys and subtrees at a node.
- How should we store key-subtree list?
  - Array? Then we must use length 7. *This wastes space.*
  - Linked list? We have overhead for list-nodes. *This wastes space.*

  It does not matter for the theoretical bound, but matters in practice.

Better idea: Design a class of binary search trees that mirrors 2-4-trees!

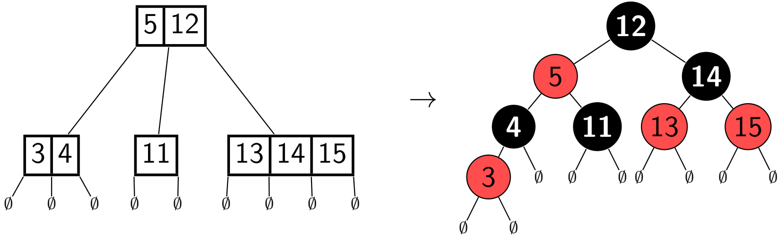
2-4-tree to red-black-tree



Converting a 2-4-tree:

- A d-node becomes a black node with $d-1$ red children
  (Assembled so that they form a BST of height at most 1.)

2-4-tree to red-black-tree



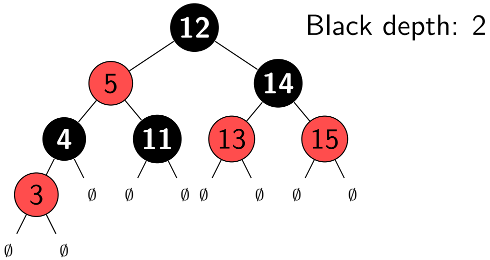
Converting a 2-4-tree:

- A d-node becomes a black node with $d-1$ red children
  (Assembled so that they form a BST of height at most 1.)

Resulting properties:

- Any red node has a black parent.
- Any empty subtree T has the same **black-depth**
  (number of black nodes on path from root to T)

Red-black-trees



Definition: A **red-black tree** is a binary search tree such that

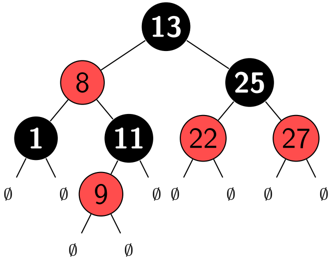
- every node has a color (red or black),
- every red node has a black parent (in particular the root is black),
- any empty subtree T has the same black-depth (number of black nodes on path from root to T)

Note: Can store this with only *one bit* overhead per node.

Red-black tree to 2-4-tree

Rather than proving properties or describing operations directly, we convert back to 2-4-trees.

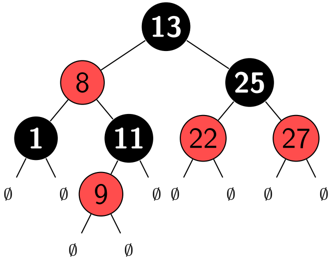
Lemma: Any red-black tree T can be converted into a 2-4-tree T'.



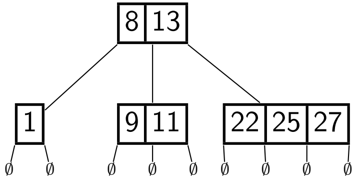
Red-black tree to 2-4-tree

Rather than proving properties or describing operations directly, we convert back to 2-4-trees.

Lemma: Any red-black tree T can be converted into a 2-4-tree T'.



$\rightarrow$



Proof:

- Black node with $0 \le d \le 2$ red children becomes a $(d+1)$-node
- This covers all nodes (no red node has a red child)
- Empty subtrees on same level due to the same blackdepth

Red-black tree summary

- Red-black trees have height $O(\log n)$.
  - Each level of the 2-4-tree creates at most 2 levels in the red-black tree.

Red-black tree summary

- Red-black trees have height $O(\log n)$.
  - Each level of the 2-4-tree creates at most 2 levels in the red-black tree.

- *insert* can be done in $O(\log n)$ worst-case time.
  - Convert relevant part to 2-4-tree.
  - Do insertion in the 2-4-tree.
  - Convert relevant parts back to red-black tree.

  It can actually be done in the red-black tree directly, using only rotations and recoloring (no details).

- *delete* can also be done in $O(\log n)$ worst-case time (no details)

Red-black tree summary

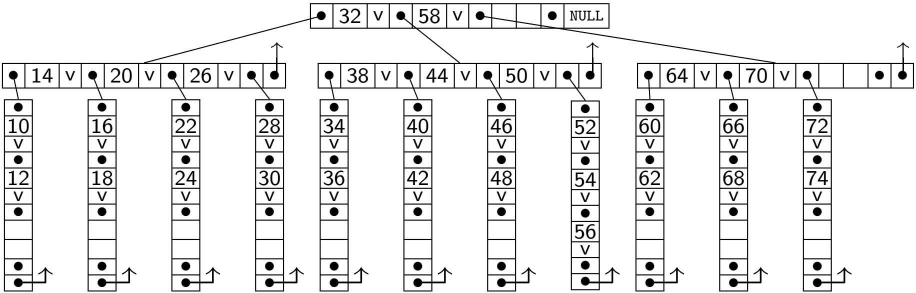
- Red-black trees have height $O(\log n)$.
  - Each level of the 2-4-tree creates at most 2 levels in the red-black tree.
- *insert* can be done in $O(\log n)$ worst-case time.
  - Convert relevant part to 2-4-tree.
  - Do insertion in the 2-4-tree.
  - Convert relevant parts back to red-black tree.

  It can actually be done in the red-black tree directly, using only rotations and recoloring (no details).
- *delete* can also be done in $O(\log n)$ worst-case time (no details)
- Experiments show that red-black tree use fewer rotations than AVL-trees.
- This is a very popular balanced binary search tree (`std::map`)

B-trees

A **B-tree** is an *a*-*b*-tree tailored to the external memory model.

- Every node is one block of memory (of size B).
- The order b is chosen maximally such that $(b-1)$-node fits into a block of memory. Typically $b \in \Theta(B)$.
- a is set to be $\lceil b/2 \rceil$ as before.



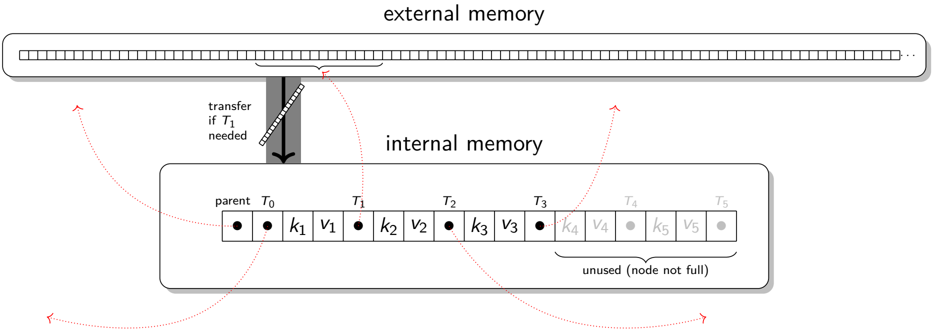
('v' indicates the value or value-reference associated with the key next to it)

(arrows indicate references to the parent)

B-tree Close-up

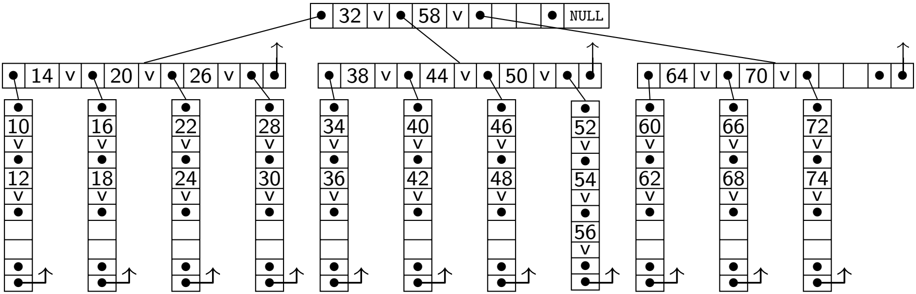
To see how to choose the order b, inspect a $(b-1)$-node:

- Stoe $b-1$ keys and $b-1$ values
- Store b references to subtrees
- Store parent-reference



In this example: $B = 17$ memory cells fit into one block, so we would choose order $b = 6$.

B-tree analysis



- *search*, *insert*, and *delete* each requires visiting $\Theta(\textit{height})$ nodes
- Work within a node is done in internal memory $\Rightarrow$ no block-transfer.
- The height is $\Theta(\log_a n) = \Theta(\log_B n)$ (since $a = \lceil b/2 \rceil \in \Theta(B)$)

So all operations require $\Theta(\log_B n)$ **block transfers**.

B-tree summary

- All operations require $\Theta(\log_B n)$ **block transfers**.
  - This is asymptotically optimal.
  - **Can show:** Searching among n items requires $\Omega(\log_B n)$ block transfers.
- In practice, height is a small constant.
  - Say $n = 2^{50}$, and $B = 2^{15}$. So roughly $b = \frac{1}{3}2^{15}$, $a = \frac{1}{3}2^{14}$.
  - B-tree of height 4 would have $\geq 2a^4 - 1 > 2^{50}$ KVPs.
  - So height is 3.
- There are some variations that are even better in practice.
- B-trees are hugely important for storing data bases ($\rightsquigarrow$ cs448)

Outline

11 External Memory

- Motivation
- Stream-based algorithms
- External Dictionaries
  - *a*-*b*-trees
  - 2-4-trees and Red-Black Trees
  - B-trees
- External Hashing

Dictionaries for Hash-values in External Memory

Recall Hashing:

- Use hash-function to map keys to (small) integers.
- Expected run-time of operations is $O(1)$ if load factor α is kept small and hash-function is chosen randomly

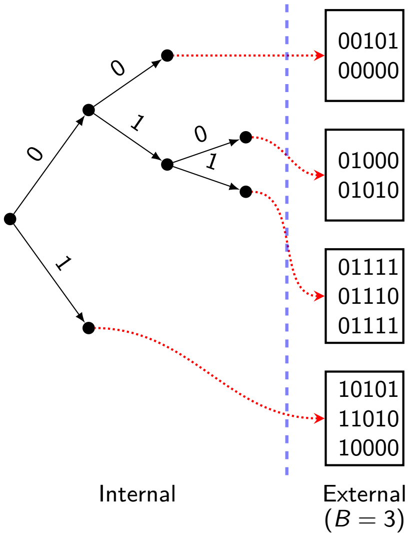
This does not adapt well to external memory.

- We must occasionally re-hash to keep load factor α small.
- And re-hashing must load *all* n/B blocks.
- This is unacceptably slow.

Goal: Data structure for hash-values that typically uses $O(1)$ block transfers, and never needs to load all blocks.

Idea: Keys $\rightsquigarrow$ Hash-values $=$ integers $\rightsquigarrow$ fixed-length bitstrings.
Store trie of bitstrings whose leaves are blocks of memory.

Trie of blocks – Overview



Assumption: We store fixed-length bitstrings.

[These come from hash-values and are not necessarily distinct.]

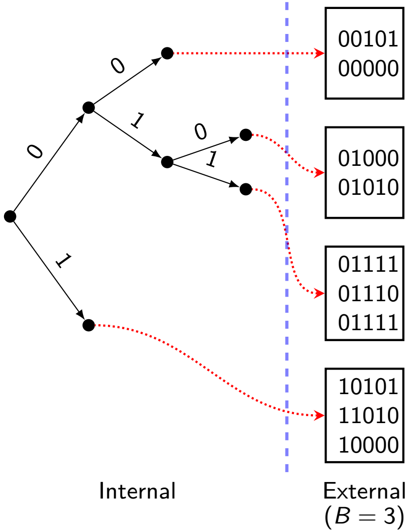
Build trie D (the **directory**) of bitstrings in internal memory.

Stop splitting in D when remaining items fit in one block.
($\sim$ pruned trie, but stop earlier)

Each leaf of D refers to a block of external memory.

The blocks store KVPs in no particular order.

Trie of blocks – operations

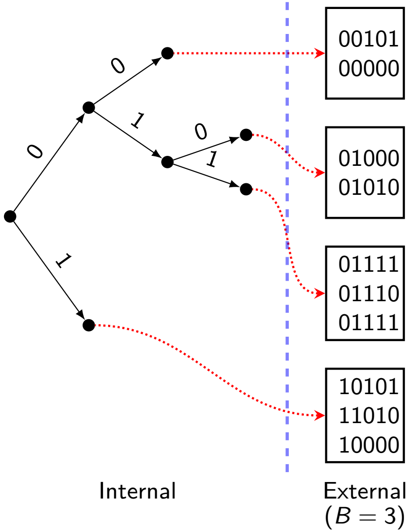


search(k):

- Search for k in D until we reach leaf ℓ
- Load block at ℓ
- Search for k in block

1 block transfer.

Trie of blocks – operations



search(k):

- Search for k in D until we reach leaf ℓ
- Load block at ℓ
- Search for k in block

1 block transfer.

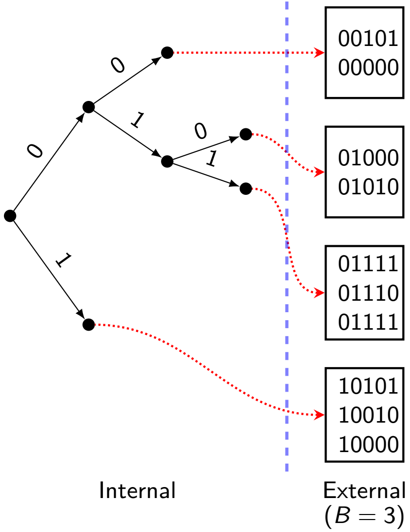
delete(k):

- *search*(k) loads block
- delete k from block
- Transfer updated block back

2 block transfers.

(Optional: combine underfull blocks. This costs block-transfers, and normally is not worth the space-savings.)

Trie of blocks – operations

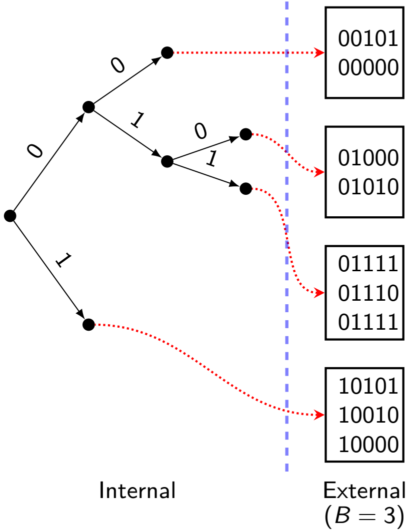


insert(k):

- Search for k in D until we reach leaf ℓ
- Load block P at ℓ
- If P is at capacity
  - Leaf ℓ gets two new children
  - Create two new blocks
  - Split items in ℓ by next bit
- Insert k into appropriate block.
- Transfer updated block back

Typically $2 - 3$ block transfers.

Trie of blocks – operations



insert(k):

- Search for k in D until we reach leaf ℓ
- Load block P at ℓ
- If P is at capacity
  - Leaf ℓ gets two new children
  - Create two new blocks
  - Split items in ℓ by next bit
- Insert k into appropriate block.
- Transfer updated block back

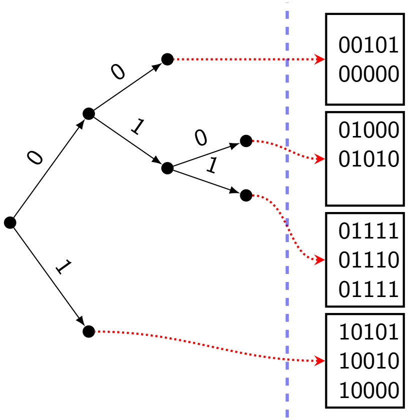
Typically $2-3$ **block transfers**.

If *all* items in P have the same next bit, then split repeatedly.

For big B, this is (extremely) unlikely.

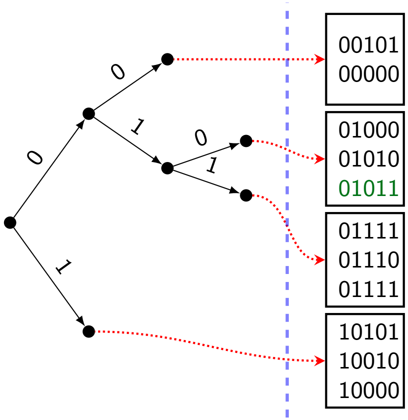
Example 1: Insert

insert(`01011`)



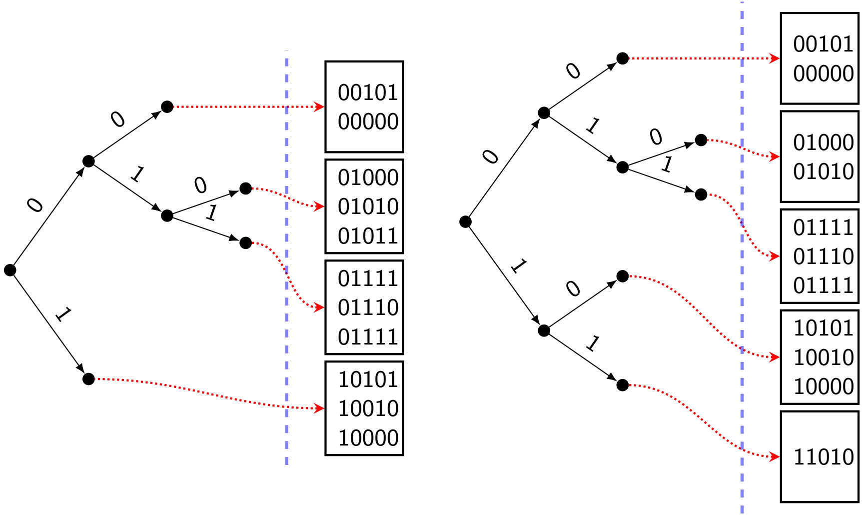
Example 1: Insert

insert(01011)



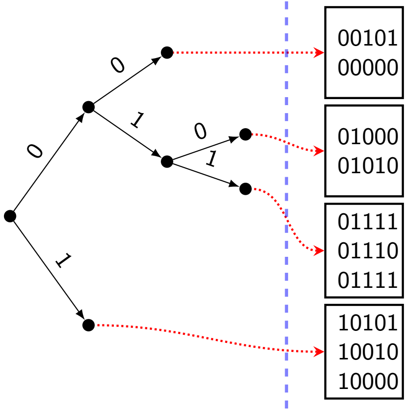
Example 1: Insert

insert(11010)



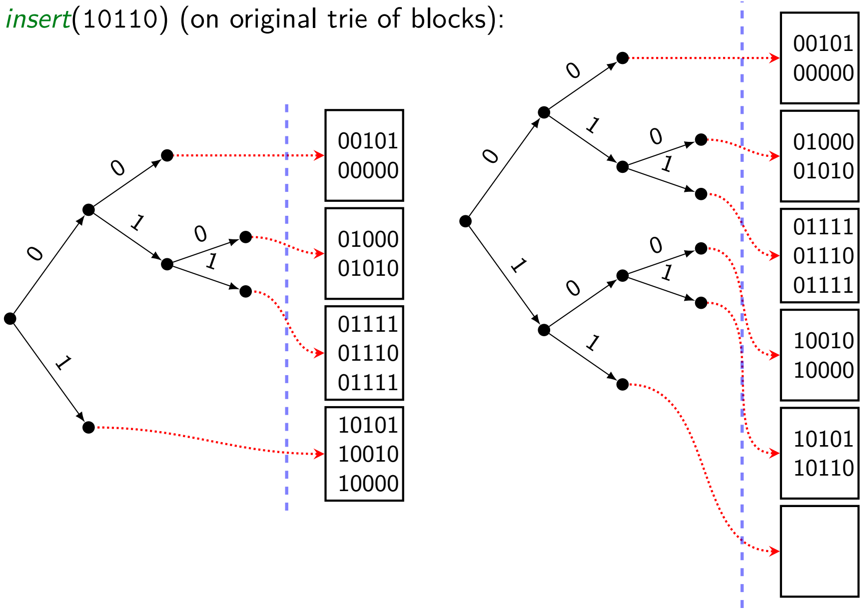
Example 2: Insert

insert(`10110`) (on original trie of blocks):



Example 2: Insert

insert(`10110`) (on original trie of blocks):



External hashing collisions

- Hashing collisions mean duplicate bitstrings, so all colliding items are in the same block.
- We do not care how collisions are resolved within the block.
- But what if more than B items have the same hash-value?
  - All bistrings in block are the same, so we cannot split
  - This means either the load factor is too big or the hash-function is bad. Either way, normally we would re-hash.

$$\xrightarrow{\textit{insert}(10010)} \boxed{\begin{array}{c} 10010 \\ 10010 \\ 10010 \end{array}}$$

External hashing collisions

- Hashing collisions mean duplicate bitstrings, so all colliding items are in the same block.
- We do not care how collisions are resolved within the block.
- But what if more than B items have the same hash-value?
  - All bistrings in block are the same, so we cannot split
  - This means either the load factor is too big or the hash-function is bad. Either way, normally we would re-hash.

$$\xrightarrow{\textit{insert}(10010)} \boxed{\begin{array}{c}10010\\ 10010\\ 10010\end{array}} \quad \text{becomes} \quad \xrightarrow{\textit{insert}(10010^\frown 010)} \boxed{\begin{array}{c}10010^\frown 000\\ 10010^\frown 010\\ 10010^\frown 110\end{array}}$$

- Here instead we *extend* the hash-function:
  Replace $h(k)$ by $h(k)^\frown h'(k)$ for some new hash-function $h'(\cdot)$.
- Initial bits are unchanged $\rightarrow$ other blocks unaffected.

External hashing summary

- Only $O(1)$ block transfers expected for *any* operation.
- To make more space, we typically only add one block.
  We rarely change the size of the directory.
  We *never* have to move all items. (in contrast to re-hashing!)

External hashing summary

- Only $O(1)$ block transfers expected for *any* operation.
- To make more space, we typically only add one block.
  We rarely change the size of the directory.
  We *never* have to move all items. (in contrast to re-hashing!)
- Directory D typically fits into in internal memory.
  - If it does not, then strategies similar to B-trees can be applied.
  - D can also be stored as an array, which typically makes it smaller (no details).
- Many blocks will not be full, but space usage is not too inefficient
  - Can show: for randomly chosen bitstrings each block is expected to be 69% full.