CS 240 – Data Structures and Data Management

Module 11: External Memory

Mark Petrick Olga Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2020

References: Goodrich & Tamassia 20.1-20.3, Sedgewick 16.4

version 2020−03−28 10:50

Outline

Outline

Different levels of memory

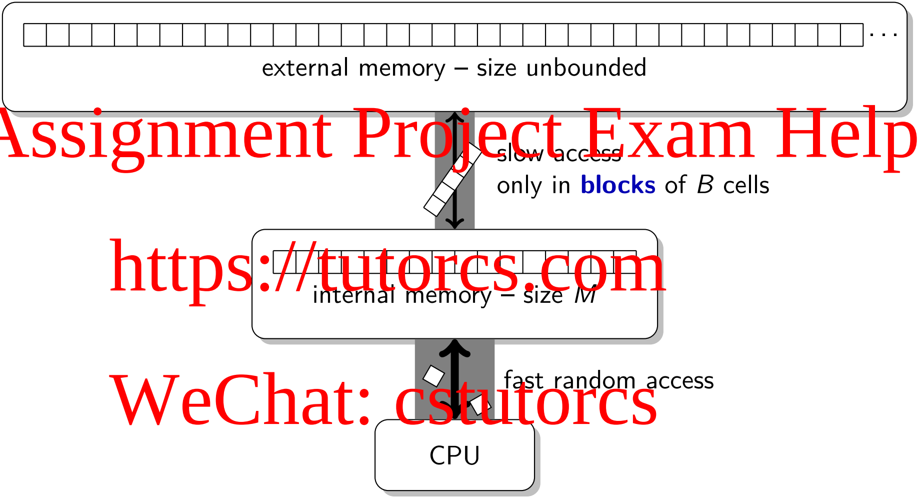
Current architectures:

- 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?

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

The External-Memory Model (EMM)



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

Outline

1. External Memory
   - Motivation
   - External sorting
   - External Dictionaries
   - 2-4 Trees
   - *a*-*b*-Trees
   - B-Trees
   - Extendible Hashing

Sorting in external memory

Recall: The sorting problem:
Given an array A of n numbers, put them into sorted order.

Now assume n is huge and A is stored in blocks in external memory.

- Heapsort was optimal in time and space in RAM model
- But: Heapsort accesses A at indices that are far apart
  $\rightsquigarrow$ typically one block transfer per array access
  $\rightsquigarrow$ typically $\Theta(n \log n)$ block transfers.
  Can we do better?

Sorting in external memory

Recall: The sorting problem:
Given an array A of n numbers, put them into sorted order.

Now assume n is huge and A is stored in blocks in external memory.

- Heapsort was optimal in time and space in RAM model
- But: Heapsort accesses A at indices that are far apart
  $\rightsquigarrow$ typically one block transfer per array access
  $\rightsquigarrow$ typically $\Theta(n \log n)$ block transfers.
  Can we do better?

- Mergesort adapts well to external memory. Recall algorithm:
  - Split input in half
  - Sort each half recursively $\rightarrow$ two sorted parts
  - Merge sorted parts.

Merge

```
Merge(S1, S2)
S1, S2 are input streams that are in sorted order
    S ← output stream
    while S1 or S2 is not empty do
        if (S1 is empty) S.append(S2.pop())
        else if (S2 is empty) S.append(S1.pop())
        else if (S1.top() < S2.top()) S.append(S1.pop())
        else S.append(S2.pop())
```


External:

S_1, sorted:

2	8	10	11	12	13	21	22	[illegible]	29	30	31	[illegible]	35	36	37	33	40	42	45	49	52	53	54

S_2, sorted:

1	3	4	5	6	7	9	14	15	16	17	18	20	23	24	26	27	32	43	44	46	47	48	50	54

S :

Internal $(B = 2, M = 8)$:

Merge

Merge(S_1, S_2)
S_1, S_2 are input streams that are in sorted order

```
S ← output stream
while S1 or S2 is not empty do
    if (S1 is empty) S.append(S2.pop())
    else if (S2 is empty) S.append(S1.pop())
    else if (S1.top() < S2.top()) S.append(S1.pop())
    else S.append(S2.pop())
```


External:

2	8	10	11	12	13	21	22	[illegible]	29	30	31	[illegible]	35	36	37	33	40	42	45	49	52	53	54

←— S_1, sorted —→

1	3	[illegible]	5	6	7	9	14	15	16	17	18	20	23	24	26	27	32	43	44	46	47	48	50	54

←— S_2, sorted —→

S :

Internal $(B = 2, M = 8)$:

2	8	1	3				

←S_1→ ←S_2→ ←output→

Merge

Merge(S_1, S_2)

S_1, S_2 are input streams that are in sorted order

1. $S \leftarrow$ output stream
2. **while** S_1 or S_2 is not empty **do**
3. **if** (S_1 is empty) $S.append(S_2.pop())$
4. **else if** (S_2 is empty) $S.append(S_1.pop())$
5. **else if** ($S_1.top() < S_2.top()$) $S.append(S_1.pop())$
6. **else** $S.append(S_2.pop())$

External:

S_1, sorted:

2	8	10	11	12	13	21	22	28	29	30	31	[illegible]	35	36	37	33	40	42	45	49	52	53	54

S_2, sorted:

1	3	4	5	6	7	9	14	15	16	17	18	20	23	24	26	27	32	43	44	46	47	48	50	54

S :

Internal ($B = 2, M = 8$):

2	8		3			1	

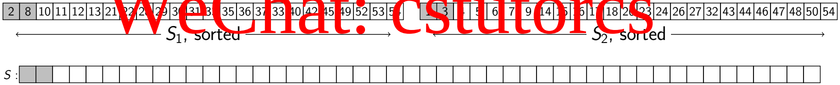
$\longleftarrow S_1 \longrightarrow$ $\longleftarrow S_2 \longrightarrow$ $\longleftarrow$output$\longrightarrow$

Merge

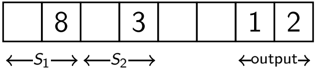
Merge(S_1, S_2)
S_1, S_2 are input streams that are in sorted order

```
S ← output stream
while S1 or S2 is not empty do
        if (S1 is empty) S.append(S2.pop())
        else if (S2 is empty) S.append(S1.pop())
        else if (S1.top() < S2.top()) S.append(S1.pop())
        else S.append(S2.pop())
```

External:



Internal ($B = 2, M = 8$):



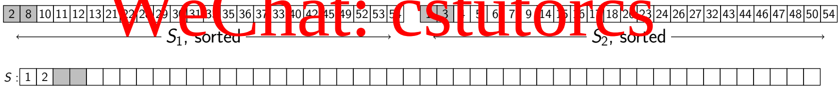
Transfer output-block to external memory when full.

Merge

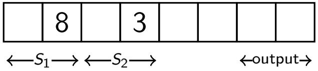
Merge(S_1, S_2)
S_1, S_2 are input streams that are in sorted order

1. $S \leftarrow$ output stream
2. **while** S_1 or S_2 is not empty **do**
3. **if** (S_1 is empty) $S.append(S_2.pop())$
4. **else if** (S_2 is empty) $S.append(S_1.pop())$
5. **else if** ($S_1.top() < S_2.top()$) $S.append(S_1.pop())$
6. **else** $S.append(S_2.pop())$

External:



Internal ($B = 2, M = 8$):



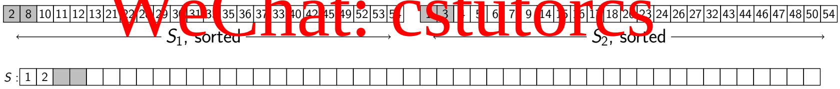
Transfer output-block to external memory when full.

Merge

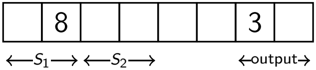
Merge(S_1, S_2)
S_1, S_2 are input streams that are in sorted order

$S \leftarrow$ output stream
while S_1 or S_2 is not empty **do**
    if (S_1 is empty) S.*append*(S_2.*pop*())
    else if (S_2 is empty) S.*append*(S_1.*pop*())
    else if (S_1.*top*() $<$ S_2.*top*()) S.*append*(S_1.*pop*())
    else S.*append*(S_2.*pop*())

External:



Internal ($B = 2, M = 8$):



Transfer output-block to external memory when full.
Load next input-block when previous is empty.

Merge

Merge(S_1, S_2)
S_1, S_2 are input streams that are in sorted order
1. $S \leftarrow$ output stream
2. **while** S_1 or S_2 is not empty **do**
3. **if** (S_1 is empty) $S.$*append*$(S_2.$*pop*$())$
4. **else if** (S_2 is empty) $S.$*append*$(S_1.$*pop*$())$
5. **else if** $(S_1.$*top*$() < S_2.$*top*$())$ $S.$*append*$(S_1.$*pop*$())$
6. **else** $S.$*append*$(S_2.$*pop*$())$

External:

S_1, sorted:

2	8	10	11	12	13	21	22	[illegible]	29	30	31	[illegible]	35	36	37	33	40	42	45	49	52	53	54

S_2, sorted:

1	3	4	5	6	7	9	14	15	16	17	18	20	23	24	26	27	32	43	44	46	47	48	50	54

S:

1	2																																															

Internal ($B = 2, M = 8$):

S_1		S_2				output	
	8	4	5			3	

Transfer output-block to external memory when full.
Load next input-block when previous is empty.

Mergesort in external memory

- *Merge* takes $O(n/B)$ block-transfers:
  - To merge, *only* need to know first item in S_1 and S_2 $\Rightarrow$ only need leftmost block from S_i $(i = 1, 2)$
  - Once the leftmost block of S_i is loaded, we need no other block of S_i until all its numbers have been parsed. $\Rightarrow$ Each block of S_i is transferred only once.
  - Each block of output is transferred back only once.
  - So each item is transferred twice $\Rightarrow \approx 2n/B$ block-transfers.
- Recall: Mergesort uses $O(\log_2 n)$ rounds of merging.

$\Rightarrow$ Mergesort uses $O(n/B \cdot \log_2 n)$ block-transfers.

Better sorting in external memory

- **Observe**: We had space left in internal memory during *Merge*.
- **Idea**: We could merge d parts at once.
- Here $d \approx M/B - 1$ so that $d+1$ blocks fit into main memory.
  - d blocks from the d input-parts
  - One block for the output.

External:

2	8	11	12	13	15	21	22	28	29	30	31	33	35	36	54

$\longleftarrow$ S_1, sorted $\longrightarrow$

6	10	17	18	20	23	24	26	33	37	40	42	45	49	52	53

$\longleftarrow$ S_2, sorted $\longrightarrow$

1	3	4	5	7	9	14	16	27	32	43	44	46	47	48	50	54

$\longleftarrow$ S_3, sorted $\longrightarrow$

S :

Internal ($B = 2, M = 8$):

Better sorting in external memory

- **Observe**: We had space left in internal memory during *Merge*.
- **Idea**: We could merge d parts at once.
- Here $d \approx M/B - 1$ so that $d+1$ blocks fit into main memory.
  - d blocks from the d input-parts
  - One block for the output.

External:

2	8	11	12	13	15	21	22	28	29	30	31	33	35	36	54

← S_1, sorted →

6	10	17	18	20	23	24	26	33	37	40	42	45	49	52	53

← S_2, sorted →

1	3	4	5	7	9	14	16	27	32	43	44	46	47	48	50	54

← S_3, sorted →

S :

Internal ($B = 2$, $M = 8$):

2	8	6	10	1	3		
←S_1→		←S_2→		←S_3→		←output→	

Better sorting in external memory

- **Observe**: We had space left in internal memory during *Merge*.
- **Idea**: We could merge d parts at once.
- Here $d \approx M/B - 1$ so that $d+1$ blocks fit into main memory.
  - d blocks from the d input-parts
  - One block for the output.

External:

2	8	11	12	13	15	21	22	28	29	30	31	33	35	36	54

← S_1, sorted →

6	10	17	18	20	23	24	26	33	37	40	42	45	49	52	53

← S_2, sorted →

1	3	4	5	7	9	14	16	27	32	43	44	46	47	48	50	54

← S_3, sorted →

S :

Internal ($B = 2, M = 8$):

2	8	6	10		3	1	

←S_1→ ←S_2→ ←S_3→ ←output→

Better sorting in external memory

- **Observe**: We had space left in internal memory during *Merge*.
- **Idea**: We could merge d parts at once.
- Here $d \approx M/B - 1$ so that $d+1$ blocks fit into main memory.
  - d blocks from the d input-parts
  - One block for the output.

External:

2	8	11	12	13	15	21	22	28	29	30	31	33	35	36	54

$\longleftarrow S_1$, sorted $\longrightarrow$

6	10	17	18	20	23	24	26	33	37	40	42	45	49	52	53

$\longleftarrow S_2$, sorted $\longrightarrow$

1	3	4	5	7	9	14	16	27	32	43	44	46	47	48	50	54

$\longleftarrow S_3$, sorted $\longrightarrow$

S :

Internal ($B = 2, M = 8$):

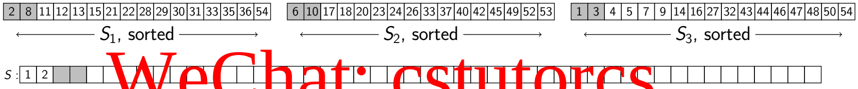
	8	6	10		3	1	2

$\longleftarrow S_1 \longrightarrow$ $\longleftarrow S_2 \longrightarrow$ $\longleftarrow S_3 \longrightarrow$ $\longleftarrow$output$\longrightarrow$

Better sorting in external memory

- **Observe**: We had space left in internal memory during *Merge*.
- **Idea**: We could merge d parts at once.
- Here $d \approx M/B - 1$ so that $d+1$ blocks fit into main memory.
  - d blocks from the d input-parts
  - One block for the output.

External:



Internal ($B = 2, M = 8$):

	8	6	10		3		

$\longleftarrow S_1 \longrightarrow$ $\longleftarrow S_2 \longrightarrow$ $\longleftarrow S_3 \longrightarrow$ $\longleftarrow$output$\longrightarrow$

Better sorting in external memory

- **Observe**: We had space left in internal memory during *Merge*.
- **Idea**: We could merge d parts at once.
- Here $d \approx M/B - 1$ so that $d+1$ blocks fit into main memory.
  - d blocks from the d input-parts
  - One block for the output.

External:

2	8	11	12	13	15	21	22	28	29	30	31	33	35	36	54

←— S_1, sorted —→

6	10	17	18	20	23	24	26	33	37	40	42	45	49	52	53

←— S_2, sorted —→

1	3	4	5	7	9	14	16	27	32	43	44	46	47	48	50	54

←— S_3, sorted —→

S :

1	2																																														

Internal ($B = 2, M = 8$):

	8	6	10			3	

←S_1→ ←S_2→ ←S_3→ ←output→

Better sorting in external memory

- **Observe**: We had space left in internal memory during *Merge*.
- **Idea**: We could merge d parts at once.
- Here $d \approx M/B - 1$ so that $d+1$ blocks fit into main memory.
  - d blocks from the d input-parts
  - One block for the output.

External:

2	8	11	12	13	15	21	22	28	29	30	31	33	35	36	54

S_1, sorted

6	10	17	18	20	23	24	26	33	37	40	42	45	49	52	53

S_2, sorted

1	3	4	5	7	9	14	16	27	32	43	44	46	47	48	50	54

S_3, sorted

S : 1 2

Internal ($B = 2, M = 8$):

	8	6	10	4	5	3	

$\longleftarrow S_1 \longrightarrow$ $\longleftarrow S_2 \longrightarrow$ $\longleftarrow S_3 \longrightarrow$ $\longleftarrow$output$\longrightarrow$

d-way merge

- When consider the *internal run-time*, d-way merge is fastest when using a *min-oriented* priority queue.


```
d-way-merge(S_1, ..., S_d)
S_1, ..., S_d are input streams that are in sorted order
    P ← empty min-priority queue
    S ← output stream
    for i ← 1 to d do P.insert( (S_i.top(), i) )
                // each item in P keeps track of its input-steam
    while P is not empty do
        (x, i) ← P.deleteMin()
        S.append(S_i.pop())
        if S_i is not empty do
            P.insert( (S_i.top(), i) )
```


- The run-time for *d-way-merge* is $O(n \log d)$.
- The number of *block transfers* is again $O(n/B)$.

d-way mergesort

Idea: Apply Mergesort but split into many parts.

- Set $d \approx M/B - 1$
- Split the input into d approximately equal parts $S_1, \ldots, S_d$.
- Recursively sort $S_1, \ldots, S_d$.
- *d-way-merge*$(S_1, \ldots, S_d)$

Run-time: $T(n) = dT(n/d) + O(n \log d)$

- The recursion-depth is $O(\log_d n)$
- On each level, we do $O(n \log d)$ work.
- Total time: $O(\log_d n \cdot n \log d) = O(n \log n)$
- In the RAM model (internal memory), this is no better than regular Mergesort.

Block-transfers: We have $O(n/B)$ block-transfers per level.

- Total # block-transfers: $O(\log_d n \cdot (n/B))$.

Mergesort with external memory

Total # block transfers with d-way mergesort: $O(\log_d(n) \cdot (n/B))$

One can prove lower bounds in the external memory model:

Any comparison-based sorting algorithm requires $\Omega(\frac{n}{B}\log_{M/B}(\frac{n}{B}))$ block transfers

(The proof is beyond the scope of the course.)

Mergesort with external memory

Total # block transfers with d-way mergesort: $O(\log_d(n) \cdot (n/B))$

One can prove lower bounds in the external memory model:

Any comparison-based sorting algorithm requires $\Omega(\frac{n}{B}\log_{M/B}(\frac{n}{B}))$ block transfers.

(The proof is beyond the scope of the course.)

- Recall: We can use $d \in \Theta(M/B)$ for Mergesort.
- d-way mergesort is optimal (up to constant factors)!

Outline

1. External Memory
   - Motivation
   - External sorting
   - External Dictionaries
   - 2-4 Trees
   - *a*-*b*-Trees
   - B-Trees
   - Extendible Hashing

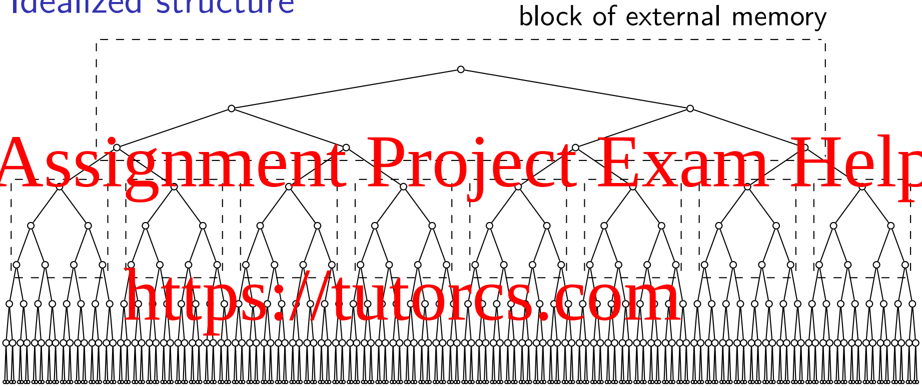
Dictionaries in external memory

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

- **Recall:** AVL-trees were optimal in time and space in RAM model
- 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

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

Idealized structure

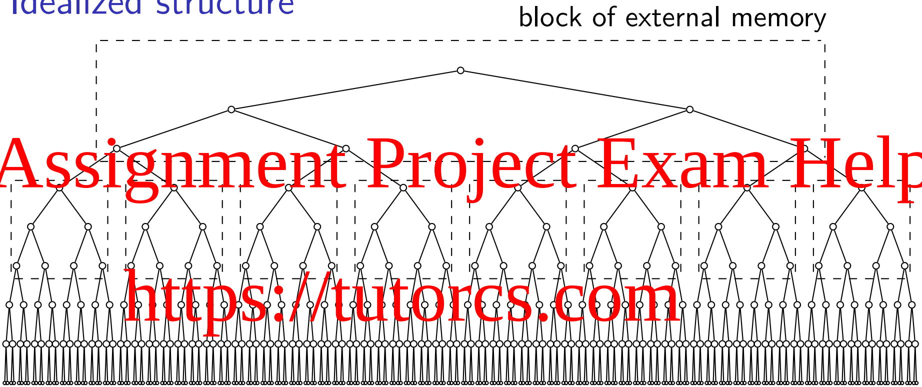


Idea: Store subtrees in one block of memory.

- If block can hold subtree of size $b-1$, then block covers height $\log b$

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

Idealized structure



Idea: Store subtrees in one block of memory.

- If block can hold subtree of size $b-1$, then block covers height $\log b$

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

- View block as one node of a *multiway-tree* ($b-1$ KVPs, b children)
- To allow *insert*/*delete*, we permit varying numbers of KVPs in nodes

Outline

1. External Memory
   - Motivation
   - External sorting

   - External Dictionaries
   - 2-4 Trees
   - *a*-*b*-Trees
   - B-Trees

   - Extendible Hashing

2-4 Trees

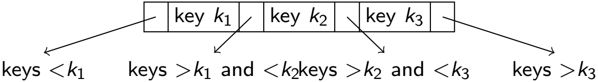
We will first explore this idea for $b = 4$. The resulting search tree is also useful for internal memory considerations.

Structural properties:

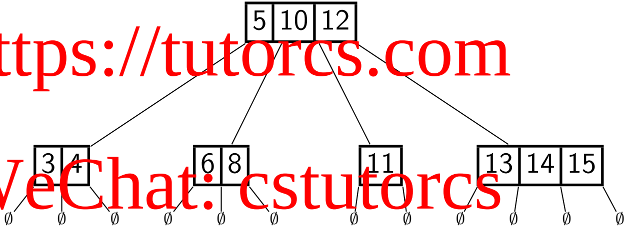
- Every node is either
  - 1-node: *one KVP* and *two subtrees* (possibly empty), or
  - 2-node: *two KVPs* and *three subtrees* (possibly empty), or
  - 3-node: *three KVPs* and *four subtrees* (possibly empty).
- All empty subtrees are at the same level.

Summary: Height-balance is strictly enforced, but allow 3 types of nodes!

Order property: The keys at a node are between the keys in the subtrees.



2-4 Tree example



2-4 Tree operations

search: The order-property determines the subtree to search in.

24Tree::search($k, v \leftarrow$ root, $p \leftarrow$ `NIL`)

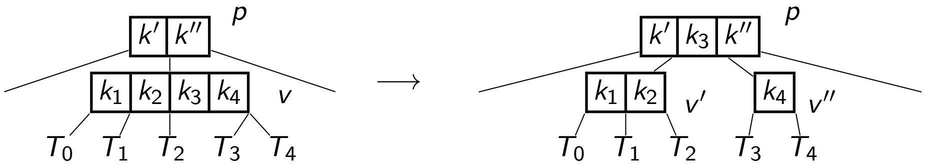
k: key to search, v: node where we search, p: parent of v

if v represents empty subtree
    return "not found, would be in p"
Let $\langle T_0, k_1, \ldots, k_d, T_d \rangle$ be key-subtree list at v
if $k \geq k_1$
    $i \leftarrow$ maximal index such that $k_i \leq k$
    if $k_i = k$
        return "at ith key in v"
    else *24Tree::search*(k, T_i, v)
else *24Tree::search*(k, T_0, v)

2-4 Tree operations

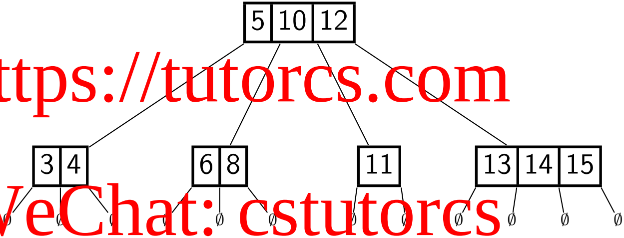
24Tree::insert(k)

$v \leftarrow$ *24Tree::search*(k) // leaf where k should be
Add k and an empty subtree in key-subtree-list of v
while v has 4 keys (**overflow** $\rightsquigarrow$ **node split**)
  Let $\langle T_0, k_1, \ldots, k_4, T_4 \rangle$ be key-subtree list at v
  if (v has no parent) create a parent of v without KVPs
  $p \leftarrow$ parent of v
  $v' \leftarrow$ new node with keys k_1, k_2 and subtrees T_0, T_1, T_2
  $v'' \leftarrow$ new node with key k_4 and subtrees T_3, T_4
  Replace $\langle v \rangle$ by $\langle v', k_3, v'' \rangle$ in key-subtree-list of p
  $v \leftarrow p$



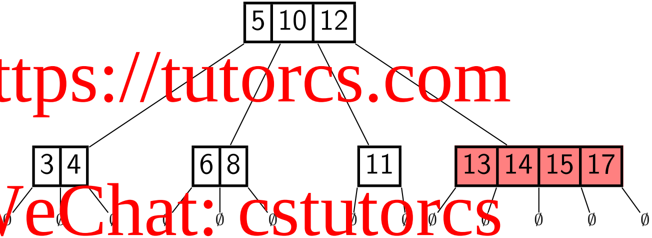
Example: Insertion in a 2-4 tree

Example: *insert*(17)



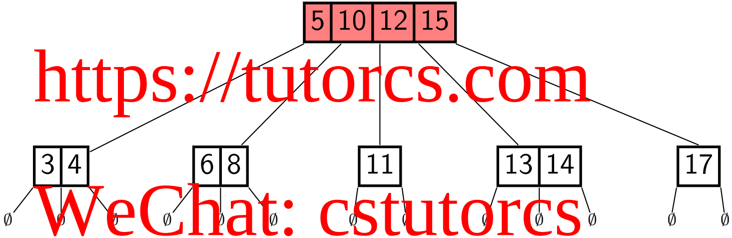
Example: Insertion in a 2-4 tree

Example: *insert*(17)



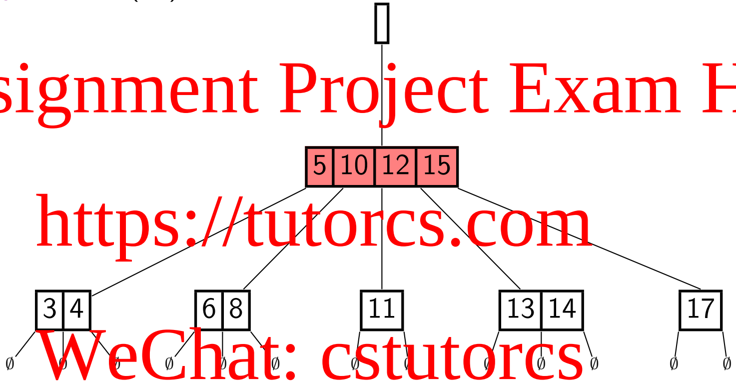
Example: Insertion in a 2-4 tree

Example: *insert*(17)



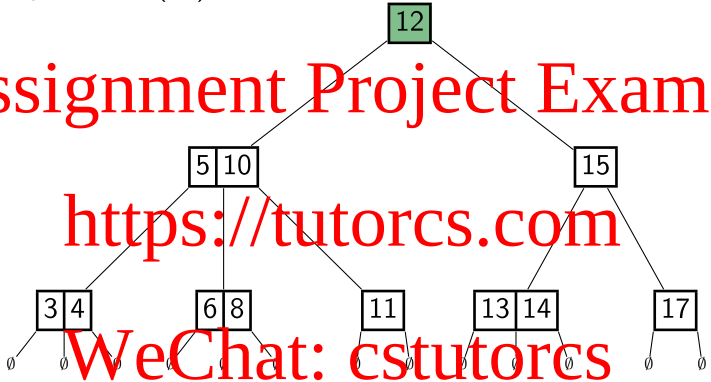
Example: Insertion in a 2-4 tree

Example: *insert*(17)



Example: Insertion in a 2-4 tree

Example: *insert*(17)



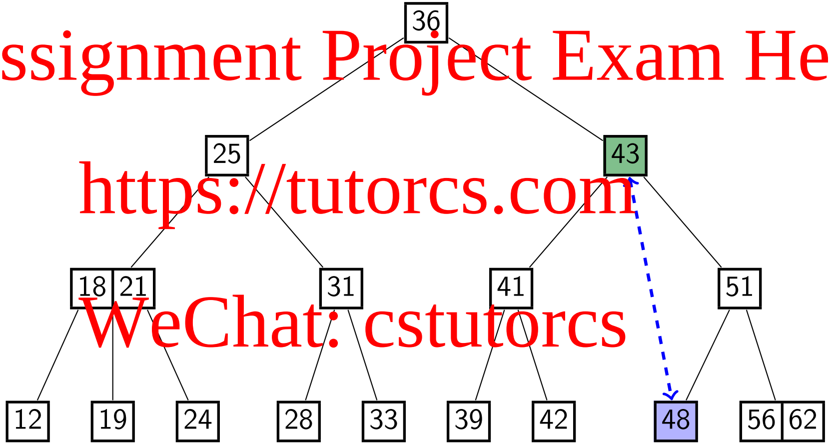
Deletion from a 2-4 Tree

24Tree::delete(k)

- $w \leftarrow$ *24Tree::search*(k) // node containing k
- $v \leftarrow$ leaf containing predecessor or successor k' of k
- Replace k by k' in w, delete k' and empty subtree in v
- **while** v has 0 keys (**underflow**)
    - **if** v is the root, delete it and **break**
    - $p \leftarrow$ parent of v
    - **if** v has a sibling u with 2 or more keys (**transfer/rotate**)
        - **if** u is right sibling
            - Replace key k in p by $u.k_1$
            - Remove $\langle u.T_0, u.k_1\rangle$ from u, append $\langle k, u.T_0\rangle$ to v
        - **else** ... // symmetrically with left sibling
    - **else** (**merge & repeat**)
        - **if** v has right sibling u
            - $v' \leftarrow$new node with list $\langle v.T_0, k, u.T_0, u.k_1, u.T_1\rangle$
            - replace $\langle v, k, u\rangle$ by $\langle v\rangle$ in p
            - $v \leftarrow p$
        - **else** ... // symmetrically with left sibling

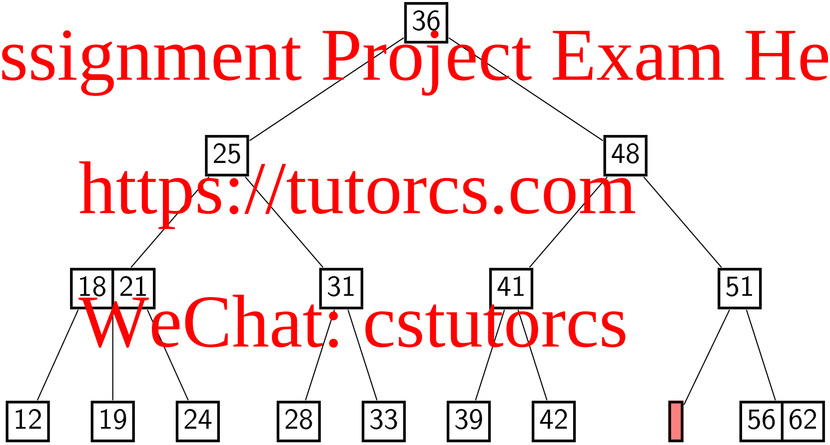
2-4 Tree Deletion

Example: *delete*(43)



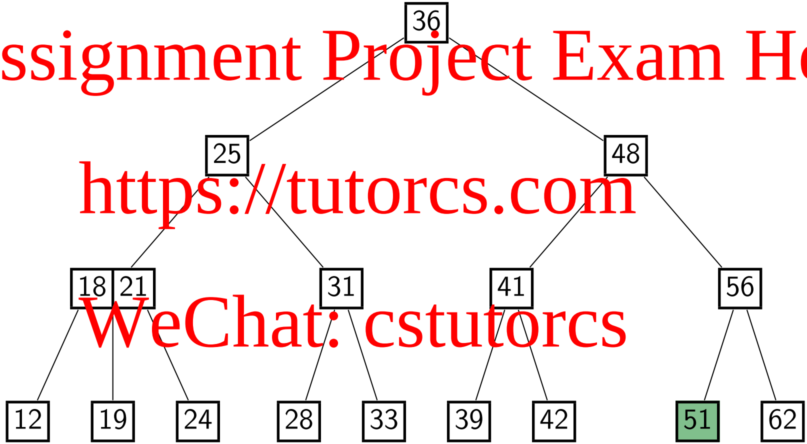
2-4 Tree Deletion

Example: *delete*(43)



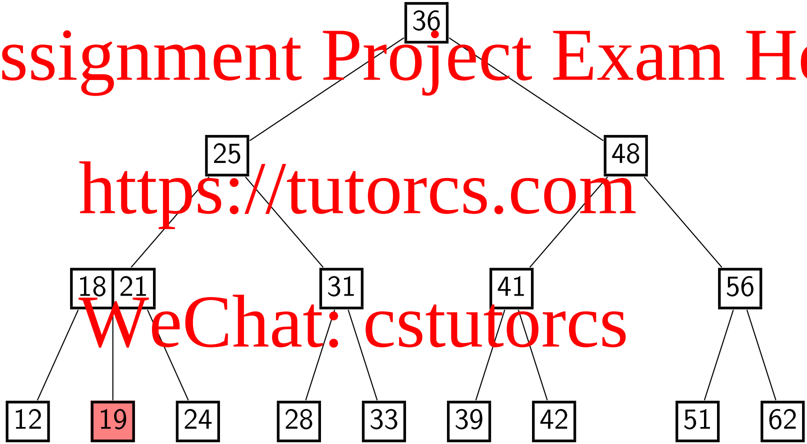
2-4 Tree Deletion

Example: *delete*(43)



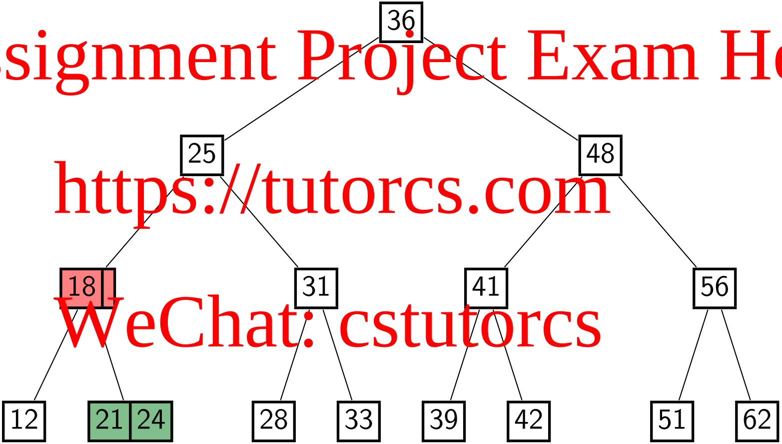
2-4 Tree Deletion

Example: *delete*(19)



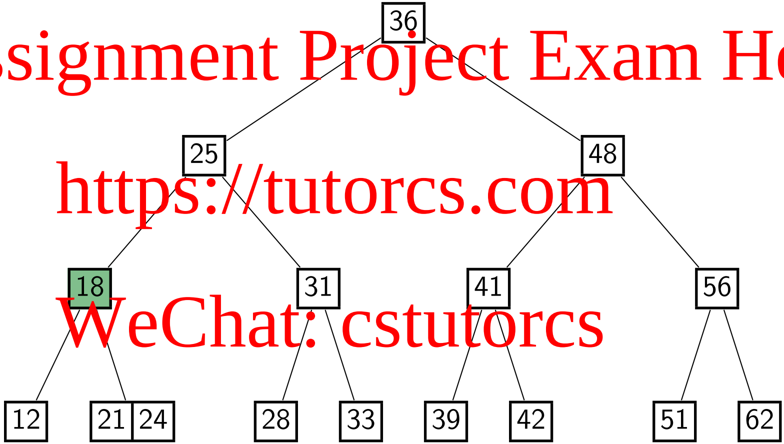
2-4 Tree Deletion

Example: *delete*(19)



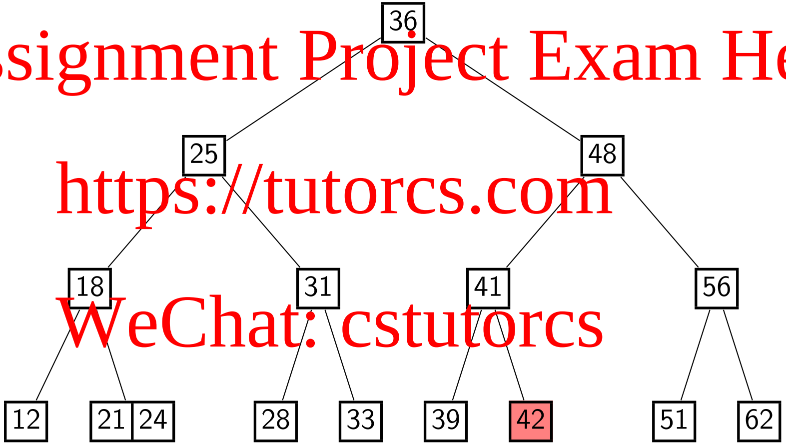
2-4 Tree Deletion

Example: *delete*(19)



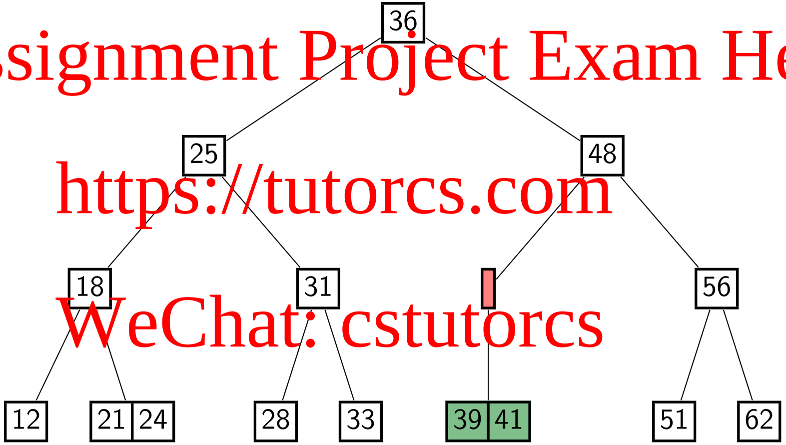
2-4 Tree Deletion

Example: *delete*(42)



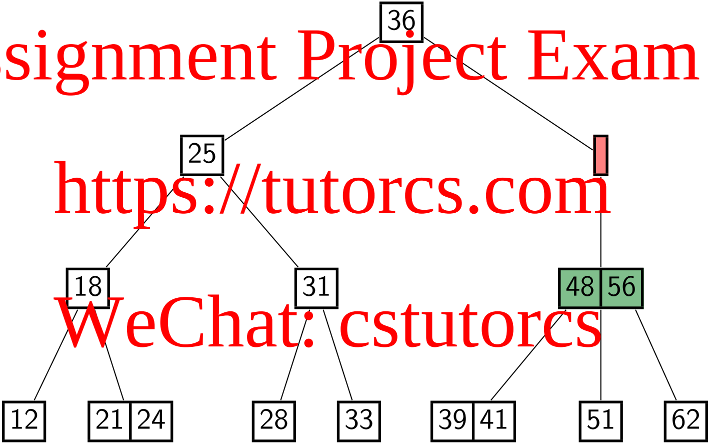
2-4 Tree Deletion

Example: *delete*(42)



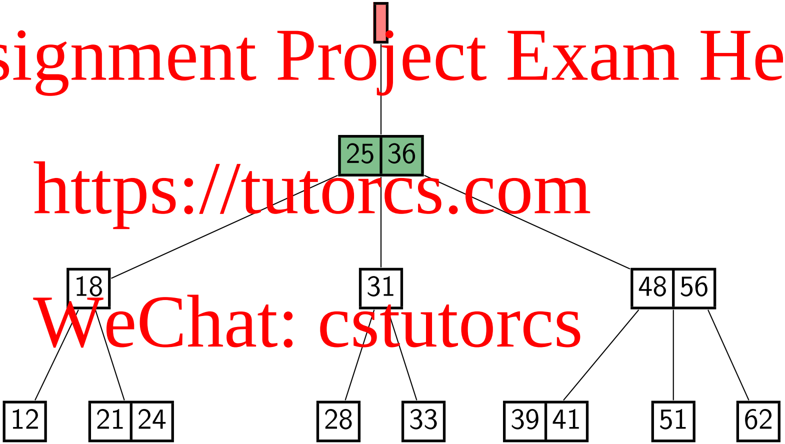
2-4 Tree Deletion

Example: *delete*(42)



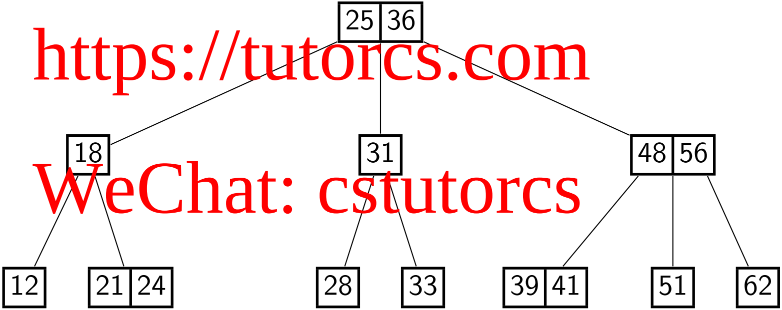
2-4 Tree Deletion

Example: *delete*(42)



2-4 Tree Deletion

Example: *delete*(42)



Outline

a-*b*-Trees

A 2-4 tree is an *a*-*b*-tree for $a = 2$ and $b = 4$.

An ***a*-*b*-tree** satisfies:

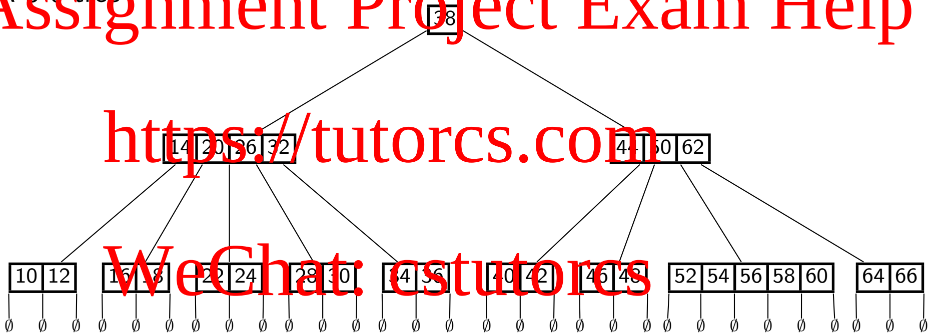
- Each node has at least a subtrees, unless it is the root.
  The root has at least 2 subtrees.
- Each node has at most b subtrees.
- If a node has k subtrees, then it stores $k-1$ key-value pairs (KVPs).
- Empty subtrees are at the same level.
- The keys in the node are between the keys in the corresponding subtrees.

Requirement: $a \leq \lceil b/2 \rceil$.

search, *insert*, *delete* then work just like for 2-4 trees, after re-defining underflow/overflow to consider the above constraints.

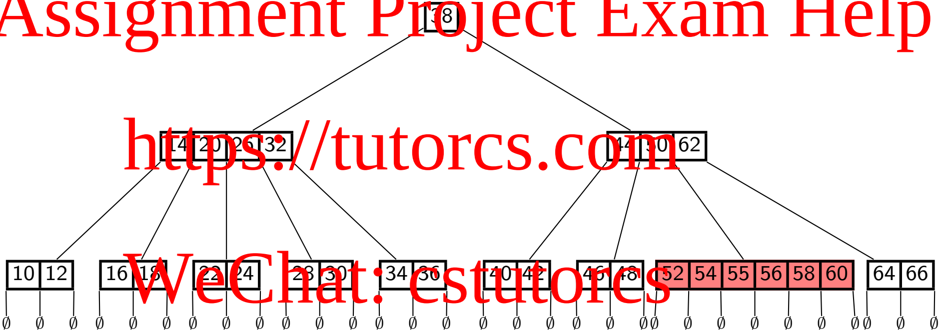
a-*b*-tree example

A 3-6 tree



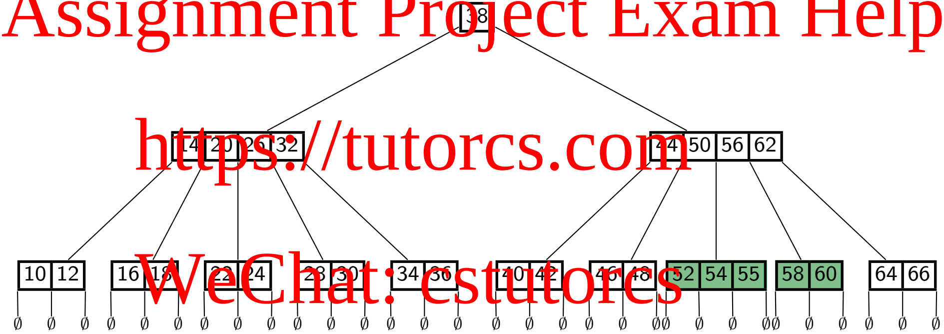
a-*b*-tree insertion

insert(55):



a-*b*-tree insertion

insert(55):



Height of an a-b-tree

Recall: $n =$ numbers of KVPs (*not* the number of nodes)

What is smallest possible number of KVPs in an a-b-tree of height-h?

Level	Nodes $\geq$	Links/node $\geq$	KVP/node $\geq$	KVPs on level $\geq$
0	1	2	1	1
1	2	a	$a-1$	$2(a-1)$
2	$2a$	a	$a-1$	$2a(a-1)$
3	$2a^2$	a	$a-1$	$2a^2(a-1)$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
h	$2a^{h-1}$	a	$a-1$	$2a^{h-1}(a-1)$

$$\text{Total: } n = \#\text{KVPs} \geq 1 + 2(a-1)\sum_{i=0}^{h-1} a^i = 2a^h - 1$$

Therefore the height of an a-b-tree is $O(\log_a(n)) = O(\log n / \log a)$.

a-b-trees as implementations of dictionaries

Analysis (if entire a-b-tree is stored in internal memory):

- *search*, *insert*, and *delete* each requires visiting $\Theta(height)$ nodes
- Height is $O(\log n / \log a)$.
- Recall: $a \le \lceil b/2 \rceil$ required for *insert* and *delete*

$\Rightarrow$ choose $a = \lceil b/2 \rceil$ to minimize the height.

- Work at node can be done in $O(\log b)$ time.

$$\text{Total cost: } O\left(\frac{\log n}{\log a} \cdot (\log b)\right) = O(\log n \cdot \frac{\log b}{\log b - 1}) = O(\log n)$$

This is no better than AVL-trees.
(Though 2-4-trees are faster than AVL-trees in practice, especially when converted to binary search trees called *red-black trees*. No details.)

The main motivation for a-b-trees is *external memory*.

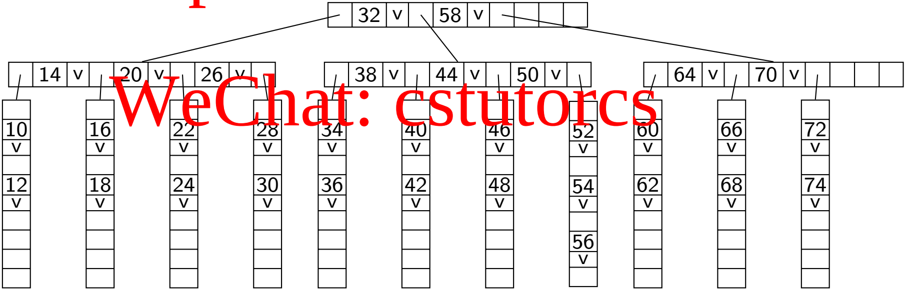
Outline

1. External Memory
   - Motivation
   - External sorting
   - External Dictionaries
   - 2-4 Trees
   - *a*-*b*-Trees
   - B-Trees
   - Extendible Hashing

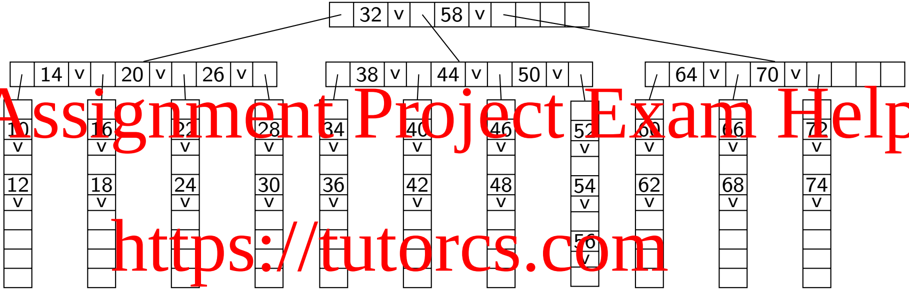
B-trees

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

- Every node is one block of memory (of size B).
- b is chosen maximally such that a node with $b-1$ KVPs (hence $b-1$ value-references and b subtree-references) fits into a block.
  b is called the **order** of the B-tree. Typically $b \in \Theta(B)$.
- a is set to be $\lceil b/2 \rceil$ as before.



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 $\Theta(\log_a n) = \Theta(\log_B n)$ (presuming $a = \lceil b/2 \rceil \in \Theta(B)$)

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

This is asymptotically optimal.

B-tree variations

For practical purposes, some variations are better:

- B-trees with **pre-emptive splitting/merging**:
  - During search for insert, split *any* node close to overflow.
  - During search for delete, merge *any* node close to underflow.

$\rightsquigarrow$ can insert/delete at leaf and stop, this halves block transfers.

- **B$^+$-trees**: All KVPs are in leaves.
  - Interior nodes store keys to guide search-path, but no values.
  - Leaves omit references to empty subtrees.

$\rightsquigarrow$ bigger order, hence smaller height.

- **Cache-oblivious** trees: What if we do not know B?
  - Build an a-b-tree with $b \approx \sqrt{n}$
  - Each node stores its KVPs again as cache-oblivious tree.

$\rightsquigarrow$ achieves $O(\log_B(n))$ block transfers *without* knowing B.

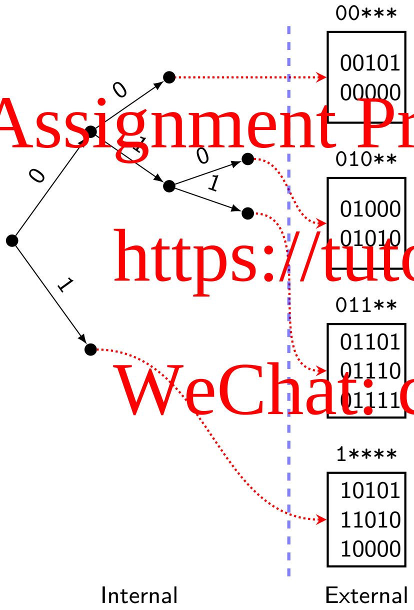
Outline

Dictionaries for Integers in External Memory

- Recall: Direct Addressing allowed for $O(1)$ insert and delete if keys are integers in $\{0, \ldots, U-1\}$
- If keys are too big, hashing was used to map keys to (smaller) integers.
- This does not adapt well to external memory.
  - Most hash strategies access many blocks (probe sequence is scattered)
  - Even those that do not (Linear Probing, Cuckoo hashing) need to re-hash to keep α small.
  - And re-hashing must load *all* blocks.
- New Idea: Store trie of links to blocks of integers.

(This is also called **extendible hashing**, because its primary use is for dictionaries that store integers that result from hashing.)

Tries of blocks – Overview



Assumption: We store integers in $\{0, 1, \ldots, 2^L - 1\}$.

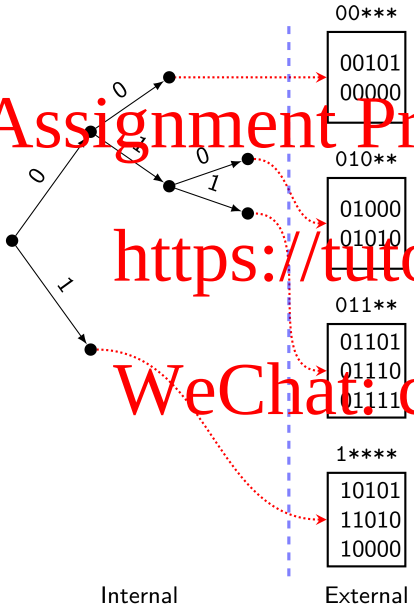
Interpret all integers as bitstrings of length L.

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

Stop splitting in trie when remaining items fit in one block.

Each leaf of D refers to block of external memory that stores the items.

External hashing with tries – Details



search(k): Search for $(k)_2$ in D until we reach leaf ℓ. Load block at ℓ and search in it.
1 block transfer.

insert(k): Search for k and load block, then insert k. If this exceeds block-capacity, split at trie-node and split blocks (possibly repeatedly).
Typically 2 block transfers.

delete(k): Search for k and load block, then use lazy deletion.
Optional: combine underfull blocks.
2 block transfers.

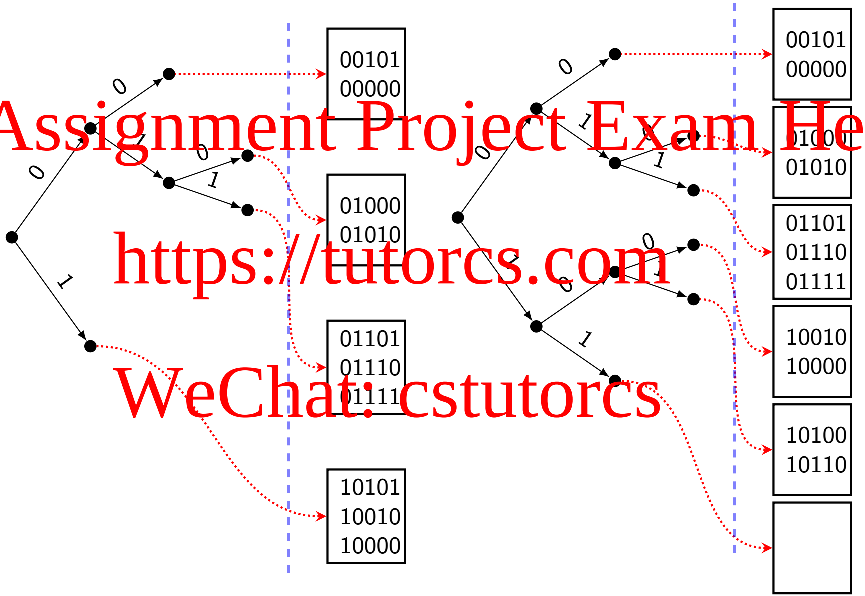
Extendible hashing: Insert

ExtendibleHashing::insert(k)
k: integer in $\{0, \ldots, 2^L - 1\}$

- Convert k to length-L bitstring w
- $\ell \leftarrow$ *Trie::search*(D, w) // leaf where w would be
- $d \leftarrow$ depth of ℓ in D
- transfer block P that ℓ refers to
- **while** P has no room for additional items
  - Split P into two blocks P_0 and P_1 by $(d{+}1)^{st}$ digit
  - Create two children ℓ_0 and ℓ_1 of ℓ, linked to P_0 and P_1
  - $d \leftarrow d{+}1$, $\ell \leftarrow \ell_{w[d]}$, $P \leftarrow P_{w[d]}$
- insert w into P

Note: This may create empty blocks, but this should be rare.

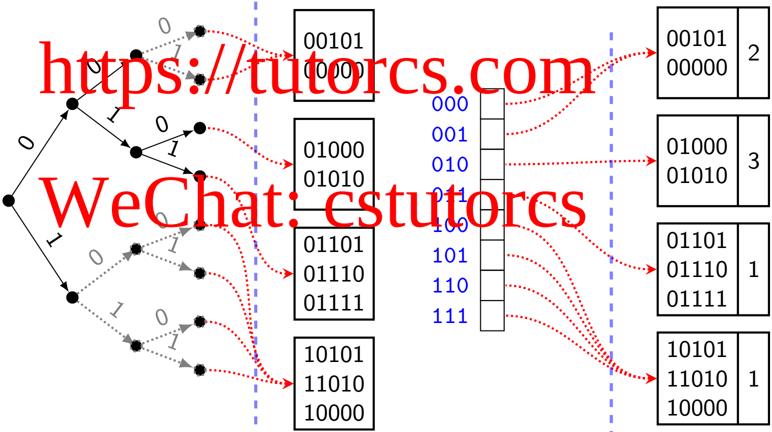
insert(10110)



Extendible hashing: saving space

We can save links (hence space in internal memory) with two tricks:

- Expand the trie so that all leaves have the same **global depth** d.
- Store *only* the leaves, and in an array D of size 2^d.
- Operations work as before if each block stores its **local depth**, i.e., the depth of the original trie-node that referred to it.



Extendible hashing discussion

- Hashing collisions (= duplicate keys) may happen, but are resolved within the block and do not affect the block transfers.
  If more items collide than can fit into a block we use other strategies (chaining, open addressing). This should be exceedingly rare.
- Directory is much smaller than total number of stored keys
  $\rightarrow$ should fit in internal memory.
  If it does not, then strategies similar to B-trees can be applied.
- Only 1 or 2 block transfers expected for *any* operation.
- To make more space, we only add one block.
  Rarely change the size of the directory.
  Never have to move all items. (in contrast to re-hashing!)
- Space usage is not too inefficient: one can show that under uniform distribution assumption each block is expected to be 69% full.