1 External Memory
- Motivation
- Stream-based algorithms
- External sorting
- External Dictionaries
- 2-4 Trees
- $a$-$b$-Trees
- B-Trees
Outline

1 External Memory
   - Motivation
     - Stream-based algorithms
     - External sorting
     - External Dictionaries
     - 2-4 Trees
     - $a$-$b$-Trees
     - B-Trees
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)

external memory – size unbounded

transfer in blocks of $B$ cells (slow)

internal memory – size $M$

random access (fast)

CPU

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

1. External Memory
   - Motivation
   - Stream-based algorithms
     - External sorting
     - External Dictionaries
     - 2-4 Trees
     - $a$-$b$-Trees
     - B-Trees
Streams and external memory

If input and output are handled via streams, then we automatically use $\Theta(\frac{n}{B})$ block transfers.

Transfer when empty

Transfer when full

$B = 5$
Streams and external memory

If input and output are handled via streams, then we automatically use $\Theta\left(\frac{n}{B}\right)$ block transfers.

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

- Pattern matching: Karp-Rabin, Knuth-Morris-Pratt, Boyer-Moore (This assumes that pattern $P$ fits into internal memory.)
- Text compression: Huffman, run-length encoding, Lempel-Ziv-Welch
Outline

1 External Memory
   • Motivation
   • Stream-based algorithms
   • External sorting
   • External Dictionaries
   • 2-4 Trees
   • a-b-Trees
   • B-Trees
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
  $\leadsto$ typically one block transfer per array access
  $\leadsto$ 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
  - $\leadsto$ typically one block transfer per array access
  - $\leadsto$ 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.

Key idea: Merge can be done with streams.
**Merge**

\[ \text{Merge}(S_1, S_2, S) \]

\( S_1, S_2 \): input streams that are in sorted order, \( S \): output stream

1. \textbf{while} \( S_1 \) or \( S_2 \) is not empty \textbf{do}
2. \hspace{1cm} \textbf{if} \ (S_1 \text{ is empty}) \ S.\text{append}(S_2.\text{pop}())
3. \hspace{1cm} \textbf{else if} \ (S_2 \text{ is empty}) \ S.\text{append}(S_1.\text{pop}())
4. \hspace{1cm} \textbf{else if} \ (S_1.\text{top}() < S_2.\text{top}()) \ S.\text{append}(S_1.\text{pop}())
5. \hspace{1cm} \textbf{else} \ S.\text{append}(S_2.\text{pop}())

\[ \Theta \left( \frac{m}{B} \right) \text{ transfers to merge} \]

Here \( B = 5 \)
Mergesort in external memory

- *Merge* uses streams $S_1, S_2, S$.
  \[ \Rightarrow \] Each block in the stream only transferred once.
- So *Merge* takes $\Theta\left(\frac{m}{B}\right)$ block-transfers to merge $m$ elements
- Recall: Mergesort uses $\lceil \log_2 n \rceil$ rounds of merging, each round merges $n$ elements
  \[ \Rightarrow \] Mergesort uses $O\left(\frac{n}{B} \cdot \log_2 n\right)$ block-transfers.

Not bad, but we can do better.

heapsort: $O\left(\frac{n}{B} \log n\right)$ block transfers
Towards \( d \)-way Mergesort

Recall: Mergesort uses \( \lceil \log_2 n \rceil \) rounds of splitting-and-merging.
Towards *d*-way Mergesort

**Observe:** We had space left in internal memory during *merge*.

- We use only three blocks, but typically $M \gg 3B$.
- **Idea:** We could merge *d* parts at once.
- Here $d \approx \frac{M}{B} - 1$ so that $d+1$ blocks fit into internal memory.
**d-way merge**

$$d\text{-}way\text{-}merge(S_1, \ldots, S_d, S)$$

- $S_1, \ldots, S_d$: input streams that are in sorted order, $S$: output stream
- 1. $P \leftarrow$ empty **min-oriented** priority queue
- 2. $\textbf{for } i \leftarrow 1 \textbf{ to } d \textbf{ do } P\text{.insert}( (S_i\text{.top},i) )$
  
  // each item in $P$ keeps track of its input-stream
- 3. $\textbf{while } P \text{ is not empty } \textbf{ do }$
- 4. $(x, i) \leftarrow P\text{.deleteMin}()$
- 5. $S\text{.append}(S_i\text{.pop})$
- 6. $\textbf{if } S_i \text{ is not empty } \textbf{ do }$
- 7. $P\text{.insert}( (S_i\text{.top},i) )$
We use a *min-oriented* priority queue $P$ to find the next item to add to the output.

- This is irrelevant for the number of block transfers.
- But there is no space-overhead needed for a priority queue. (Recall: heaps are typically implemented as arrays.)
- And with this the run-time (in RAM-model) is $O(n \log d)$.

The items in $P$ store not only the next key but also the index of the stream that contained the item.

- With this, can efficiently find the stream to reload from.

We assume $d$ is such that $d + 1$ blocks and $P$ fit into main memory.

The number of *block transfers* then is again $O\left(\frac{n}{B}\right)$. 

\[ d \leq \frac{M}{B} \ll M \]
We use a *min-oriented* priority queue $P$ to find the next item to add to the output.

- This is irrelevant for the number of block transfers.
- But there is no space-overhead needed for a priority queue. (Recall: heaps are typically implemented as arrays.)
- And with this the run-time (in RAM-model) is $O(n \log d)$.

The items in $P$ store not only the next key but also the index of the stream that contained the item.

- With this, can efficiently find the stream to reload from.

We assume $d$ is such that $d + 1$ blocks and $P$ fit into main memory.

The number of *block transfers* then is again $O\left(\frac{n}{B}\right)$.

How does *d-way merge* help to improve external sorting?
Towards $d$-way Mergesort

Recall: Mergesort uses $\lfloor \log_2 n \rfloor$ rounds of splitting-and-merging.
Towards $d$-way Mergesort

**Observe:** If we split and merge $d$-ways, there are fewer rounds.

- Number of rounds is now $\lceil \log_d n \rceil$
- We choose $d$ such that each round uses $\Theta\left(\frac{n}{B}\right)$ block transfers. (Then the number of block transfers is $\Theta(\log_d n \cdot \frac{n}{B})$.)
- Two further improvements:
  - Proceed bottom-up (while-loops) rather than top-down (recursions).
  - Save more rounds by starting immediately with runs of length $M$. 

![Diagram](image-url)
d-way mergesort

External \((B = 2)\):

\[
\begin{array}{cccccccccccccccccccc}
\end{array}
\]

Internal \((M = 8)\):

\[
\begin{array}{cccccccc}
\text{\ } & \text{\ } & \text{\ } & \text{\ } & \text{\ } & \text{\ } & \text{\ } & \text{\ }
\end{array}
\]

1. Create \(\frac{n}{M}\) sorted runs of length \(M\).
d-way mergesort

External ($B = 2$):

| 39 | 5 | 28 | 22 | 10 | 33 | 29 | 37 | 8 | 30 | 54 | 40 | 31 | 52 | 21 | 45 | 35 | 11 | 42 | 53 | 13 | 12 | 49 | 36 | 4 | 14 | 27 | 9 | 44 | 3 | 32 | 15 | 43 | 2 | 17 | 6 | 46 | 23 | 20 | 1 | 24 | 7 | 18 | 47 | 26 | 16 | 48 | 50 |

Internal ($M = 8$):

| 39 | 5 | 28 | 22 | 10 | 33 | 29 | 37 |

1. Create $\frac{n}{M}$ sorted runs of length $M$. 
d-way mergesort

External ($B = 2$):

```
39 5 28 22 10 33 29 37 8 30 54 40 31 52 21 45 35 11 42 53 13 12 49 36 4 14 27 9 44 3 32 15 43 2 17 6 46 23 20 1 24 7 18 47 26 16 48 50
```

Internal ($M = 8$):

```
5 10 22 28 29 33 37 39
```

1. Create $\frac{n}{M}$ sorted runs of length $M$. 

d-way mergesort

External \((B = 2)\):

\[
\begin{array}{cccccccccccccccccccc}
\end{array}
\]

\(\text{sorted run}\)

Internal \((M = 8)\):

\[
\begin{array}{cccccccc}
\quad & \quad & \quad & \quad & \quad & \quad & \quad & \quad
\end{array}
\]

1. Create \(\frac{n}{M}\) sorted runs of length \(M\).
d-way mergesort

External ($B = 2$):

External ($B = 2$):

Internal ($M = 8$):

1. Create $\frac{n}{M}$ sorted runs of length $M$. $\Theta(\frac{n}{B})$ block transfers
**d-way mergesort**

**External** \((B = 2)\):

```
5 10 22 28 29 33 37 51 8 21 30 31 40 45 52 53 3 4 9 14 15 27 32 44 1 2 6 17 20 23 43 46 7 16 18 24 26 47 48 50
```

**Internal** \((M = 8)\):

```
5 10 8 21 11 12
```

(priority queue not shown)

1. Create \(\frac{n}{M}\) sorted runs of length \(M\). \(\Theta\left(\frac{n}{B}\right)\) block transfers
2. Merge the first \(d \approx \frac{M}{B} - 1\) sorted runs using **d-Way-Merge**

\[
d = \frac{n}{B} - 1 = \frac{8}{2} - 1 = 3
\]
d-way mergesort

External ($B = 2$):

```
5 10 22 28 29 33 37 39 8 21 30 31 40 45 52 54 11 12 13 35 36 42 49 53 3 4 9 14 15 27 32 44 1 2 6 17 20 23 43 46 7 16 18 24 26 47 48 50
```

Internal ($M = 8$):

```
<table>
<thead>
<tr>
<th>10</th>
<th>8</th>
<th>21</th>
<th>11</th>
<th>12</th>
<th>5</th>
</tr>
</thead>
</table>
```

(priority queue not shown)

1. Create $\frac{n}{M}$ sorted runs of length $M$. $\Theta\left(\frac{n}{B}\right)$ block transfers

2. Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using d-Way-Merge
d-way mergesort

External ($B = 2$):

![Sorted runs diagram]

Internal ($M = 8$):

![Priority queue diagram]

1. Create $\frac{n}{M}$ sorted runs of length $M$. $\Theta(\frac{n}{B})$ block transfers
2. Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using d-Way-Merge
d-way mergesort

External ($B = 2$):

\[5 \ 10 \ 22 \ 28 \ 29 \ 33 \ 37 \ 39 \ 8 \ 21 \ 30 \ 31 \ 40 \ 45 \ 52 \ 54\ 11 \ 12 \ 13 \ 35 \ 36 \ 42 \ 49 \ 53 \ 3 \ 4 \ 9 \ 14 \ 15 \ 27 \ 32 \ 44 \ 1 \ 2 \ 6 \ 17 \ 20 \ 23 \ 43 \ 46 \ 7 \ 16 \ 18 \ 24 \ 26 \ 47 \ 48 \ 50\]

\[\text{sorted run} \quad \text{sorted run} \quad \text{sorted run} \quad \text{sorted run} \quad \text{sorted run} \quad \text{sorted run} \quad \text{sorted run} \quad \text{sorted run}\]

\[5 \ 8 \]

Internal ($M = 8$):

\[\begin{array}{ccccccc}
S_1 & S_2 & S_3 & S \\
10 & 21 & 11 & 12 & \text{(priority queue not shown)}
\end{array}\]

1. Create $\frac{n}{M}$ sorted runs of length $M$. $\Theta\left( \frac{n}{B} \right)$ block transfers
2. Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using $d$-Way-Merge
d-way mergesort

External ($B = 2$):

```
5 10 22 28 29 33 37 39 8 21 30 31 40 45 52 54 11 12 13 35 36 42 49 53
```

(sorted run) (sorted run) (sorted run) (sorted run) (sorted run) (sorted run) (sorted run)

5 8

Internal ($M = 8$):

```
s_{1} s_{2} \boxed{21} s_{3} 11 12 10 s
```

(priority queue not shown)

1. Create $\frac{n}{M}$ sorted runs of length $M$. $\Theta(\frac{n}{B})$ block transfers
2. Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using d-Way-Merge
d-way mergesort

External \((B = 2)\):

\[
\begin{array}{cccccccccccccccccccccccc}
\text{sorted run} & \text{sorted run} & \text{sorted run} & \text{sorted run} & \text{sorted run} & \text{sorted run} & \text{sorted run} & \text{sorted run} \\
\end{array}
\]

Internal \((M = 8)\):

\[
\begin{array}{cccc}
22 & 28 & 21 & 11 & 12 & 10
\end{array}
\]

(priority queue not shown)

1. Create \(\frac{n}{M}\) sorted runs of length \(M\). \(\Theta\left(\frac{n}{B}\right)\) block transfers

2. Merge the first \(d \approx \frac{M}{B} - 1\) sorted runs using \textit{d-Way-Merge}
d-way mergesort

External ($B = 2$):

\[
\begin{array}{cccccccccccccccccccccccc}
\end{array}
\]

\[\text{sorted run} \quad \text{sorted run} \quad \text{sorted run} \quad \text{sorted run} \quad \text{sorted run} \quad \text{sorted run} \quad \text{sorted run}\]

Internal ($M = 8$):

\[
\begin{array}{cccccccc}
22 & 28 & \_ & 21 & \_ & 12 & 10 & 11
\end{array}
\]

(priority queue not shown)

\[s_1 \quad s_2 \quad s_3 \quad s\]

1. Create $\frac{n}{M}$ sorted runs of length $M$. $\Theta\left(\frac{n}{B}\right)$ block transfers

2. Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using $d$-Way-Merge
d-way mergesort

External \((B = 2)\):

```
| 22 | 28 | 29 | 33 | 37 | 39 | 8 | 21 | 30 | 31 | 40 | 45 | 52 | 54 | 11 | 12 | 13 | 35 | 36 | 42 | 49 | 53 |
```

sorted run \rightarrow \leftarrow sorted run \rightarrow \leftarrow sorted run \rightarrow \leftarrow sorted run \rightarrow \leftarrow sorted run

\[ 5 \ 8 \ 10 \ 11 \]

Internal \((M = 8)\):

```
22 | 28 | | 21 | | 12 | | (priority queue not shown)
```

\(s_1 \quad s_2 \quad s_3 \quad s\)

1. Create \(\frac{n}{M}\) sorted runs of length \(M\). \(\Theta\left(\frac{n}{B}\right)\) block transfers
2. Merge the first \(d \approx \frac{M}{B} - 1\) sorted runs using \textit{d-Way-Merge}
d-way mergesort

External ($B = 2$):

\[
\begin{array}{c}
\end{array}
\]

(sorted run) \hspace{1cm} \text{sorted run} \hspace{1cm} \text{sorted run}

Internal ($M = 8$):

\[
\begin{array}{c}
\end{array}
\]

(priority queue not shown)

\[
\begin{array}{c}
\text{s}_1 \hspace{1cm} \text{s}_2 \hspace{1cm} \text{s}_3 \hspace{1cm} \text{s}
\end{array}
\]

1. Create $\frac{n}{M}$ sorted runs of length $M$. $\Theta\left(\frac{n}{B}\right)$ block transfers
2. Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using $d$-Way-Merge
d-way mergesort

External ($B = 2$):

```
5 8 10 11 12 13 21 22 28 29 30 31 33 35 36 37 39 40 42 45 49 52 53 54 1 2 3 4 6 7 9 14 15 16 17 18 20 23 24 26 27 32 43 44 46 47 48 50
```

Internal ($M = 8$):

```
S_1  S_2  S_3  S
```

(priority queue not shown)

1. Create $\frac{n}{M}$ sorted runs of length $M$. $\Theta(\frac{n}{B})$ block transfers
2. Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using $d$-Way-Merge
3. Keep merging the next runs to reduce $\#$ runs by factor of $d$ $\leadsto$ one round of merging. $\Theta(\frac{n}{B})$ block transfers
d-way mergesort

External ($B = 2$):

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 20 21 22 23 24 26 27 28 29 30 31 32 33 35 36 37 39 40 42 43 44 45 46 47 48 49 50 52 53 54
```

sorted run

```
5 8 10 11 12 13 21 22 28 29 30 31 33 35 36 37 39 40 42 45 49 52 53 54 1 2 3 4 6 7 9 14 15 16 17 18 20 23 24 26 27 32 43 44 46 47 48 50
```

sorted run

Internal ($M = 8$):

```
S_1  S_2  S_3  S
```

(priority queue not shown)

1. Create $\frac{n}{M}$ sorted runs of length $M$. $\Theta(\frac{n}{B})$ block transfers
2. Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using d-Way-Merge
3. Keep merging the next runs to reduce the number of runs by factor of $d$ one round of merging. $\Theta(\frac{n}{B})$ block transfers
4. Keep doing rounds until only one run is left ✓
d-way mergesort

- We have $\log_d\left(\frac{n}{M}\right)$ rounds of merging:
  - $\frac{n}{M}$ runs after initialization.
  - $\frac{n}{M}/d$ runs after one round.
  - $\frac{n}{M}/d^k$ runs after $k$ rounds $\Rightarrow k \leq \log_d\left(\frac{n}{M}\right)$.

\[ \frac{n}{M} \cdot d^k = 1 \]
\[ \Leftrightarrow d^k = \log\left(\frac{n}{M}\right) \]
\[ \Leftrightarrow k = \log_d\left(\frac{n}{M}\right) \]
d-way mergesort

- We have $\log_d\left(\frac{n}{M}\right)$ rounds of merging:
  - $\frac{n}{M}$ runs after initialization $\rightarrow O\left(\frac{n}{b}\right)$
  - $\frac{n}{M} / d$ runs after one round.
  - $\frac{n}{M} / d^k$ runs after $k$ rounds $\Rightarrow k \leq \log_d\left(\frac{n}{M}\right)$.

- We have $O\left(\frac{n}{B}\right)$ block-transfers per round.

- $d \approx \frac{M}{B} - 1$.

$\Rightarrow$ Total # block transfers is proportional to

$$\log_d\left(\frac{n}{M}\right) \cdot \frac{n}{B} \in O\left(\frac{n}{B}\right) \cdot \frac{n}{B}$$

$$\Rightarrow d \approx \frac{M}{B} - 1$$

| Algorithm       | Complexity
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>heapsort</td>
<td>$n \log(n)$</td>
</tr>
<tr>
<td>mergesort</td>
<td>$\frac{n}{B} \log(n)$</td>
</tr>
<tr>
<td>d-way mergesort</td>
<td>$\frac{n}{B} \log_d(n)$</td>
</tr>
<tr>
<td>optimized d-way</td>
<td>$\frac{n}{B} \log_2(n) / n$</td>
</tr>
</tbody>
</table>
d-way mergesort

- We have $\log_d\left(\frac{n}{M}\right)$ rounds of merging:
  - $\frac{n}{M}$ runs after initialization
  - $\frac{n}{M}/d$ runs after one round.
  - $\frac{n}{M}/d^k$ runs after $k$ rounds \(\Rightarrow k \leq \log_d\left(\frac{n}{M}\right)\).

- We have $O\left(\frac{n}{B}\right)$ block-transfers per round.

- $d \approx \frac{M}{B} - 1$.

\[\Rightarrow\] Total \# block transfers is proportional to

$$\log_d\left(\frac{n}{M}\right) \cdot \frac{n}{B} \in O\left(\log_{M/B}\left(\frac{n}{M}\right) \cdot \frac{n}{B}\right)$$

One can prove lower bounds in the external memory model:

We **require** $\Omega\left(\log_{M/B}\left(\frac{n}{M}\right) \cdot \frac{n}{B}\right)$ block transfers in any comparison-based sorting algorithm.

(The proof is beyond the scope of the course.)
d-way mergesort

- We have $\log_d\left(\frac{n}{M}\right)$ rounds of merging:
  - $\frac{n}{M}$ runs after initialization
  - $\frac{n}{M}/d$ runs after one round.
  - $\frac{n}{M}/d^k$ runs after $k$ rounds $\Rightarrow k \leq \log_d\left(\frac{n}{M}\right)$.
- We have $O\left(\frac{n}{B}\right)$ block-transfers per round.
- $d \approx \frac{M}{B} - 1$.

$\Rightarrow$ Total # block transfers is proportional to

$$\log_d\left(\frac{n}{M}\right) \cdot \frac{n}{B} \in O\left(\log_{M/B}\left(\frac{n}{M}\right) \cdot \frac{n}{B}\right)$$

One can prove lower bounds in the external memory model:

**We require** $\Omega\left(\log_{M/B}\left(\frac{n}{M}\right) \cdot \frac{n}{B}\right)$ block transfers in any comparison-based sorting algorithm.

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

- $d$-way mergesort is optimal (up to constant factors)!
Outline

1. External Memory
   - Motivation
   - Stream-based algorithms
   - External sorting
   - External Dictionaries
     - 2-4 Trees
     - a-b-Trees
     - B-Trees
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.
  \( \leadsto \) nearby nodes are unlikely to be on the same block
  \( \leadsto \) typically \( \Theta(\log n) \) block transfers per operation
- We would like to have *fewer* block transfers.

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

- Block acts as one node of a multiway-tree ($b-1$ KVPs, $b$ subtrees)
Towards $B$-trees

- **Idea:** Define *multiway-tree*
  - One node stores many KVPs
  - Always true: $b - 1$ KVPs $\Leftrightarrow$ $b$ subtrees
  - To allow *insert/delete*, we permit varying numbers of KVPs in nodes
    - This gives much smaller height than for AVL-trees
      - $\Rightarrow$ fewer block transfers

- Study first one special case: *2-4-trees*
  - Also useful for dictionaries in internal memory
  - May be faster than AVL-trees even in internal memory
Outline

1 External Memory
   - Motivation
   - Stream-based algorithms
   - External sorting
   - External Dictionaries

2-4 Trees
   - $a$-$b$-Trees
   - B-Trees
2-4 Trees

**Structural property:** 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).

**Order property:** The keys at a node are between the keys in the subtrees.
- With this, search is much like in binary search trees.

**Another structural property:** All empty subtrees are at the same level.
- This is important to ensure small height.
2-4 Tree example

- Empty trees do not count towards height
  - This tree has height 1
- Easy to show: Height is in $O(\log n)$, where $n = \# \text{ KVPs.}$
  - Layer $i$ has at least $2^i$ nodes for $i = 0, \ldots, h$
  - Each node has at least one KVP.

- Level 0: 1 node
- Level 1: $\geq 2$ nodes
- Level 2: $\geq 4$ nodes

At level $h$, we have at least $2^h$ keys.

$n \geq 2^h$

$\log(n) \geq h$
2-4 Tree Operations

- Search is similar to BST:
  - Compare search-key to keys at node
  - If not found, recurse in appropriate subtree

**Example:** $\text{search}(15)$

```
5 9 12

3 4

6 8

11

13 14 16

∅ ∅ ∅

∅ ∅ ∅

∅ ∅ ∅

∅ ∅ ∅
```
2-4 Tree Operations

- **Search is similar to BST:**
  - Compare search-key to keys at node
  - If not found, recurse in appropriate subtree

**Example:** *search(15)*
2-4 Tree Operations

- Search is similar to BST:
  - Compare search-key to keys at node
  - If not found, recurse in appropriate subtree

**Example:** $\text{search(15)}$ *not found*
2-4 Tree operations

\(24\text{Tree::search}(k, v \leftarrow \text{root}, p \leftarrow \text{NIL})\)

- **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 key-value pair at \(k_i\)
- Else \(24\text{Tree::search}(k, T_i, v)\)
- Else \(24\text{Tree::search}(k, T_0, v)\)
Insertion in a 2-4 tree

**Example:** \texttt{insert}(10)

- Do \texttt{Tree::search} and add key and empty subtree at leaf.
Insertion in a 2-4 tree

**Example:** \texttt{insert(10)}

- Do \texttt{24Tree::search} and add key and empty subtree at leaf.
- If the leaf had room then we are done.
**Example: insert(17)**
- Do `24Tree::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**.
Insertion in a 2-4 tree

**Example:** `insert(17)`
- Do `24Tree::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**.
Insertion in a 2-4 tree

Example: $\text{insert}(17)$

- Do $\text{24Tree::search}$ 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**.
Insertion in a 2-4 tree

Example: \textit{insert}(17)
- Do \texttt{24Tree::search} and add key and empty subtree at leaf.
- If the leaf had room then we are done.
- Else \textbf{overflow}: More keys/subtrees than permitted.
- Resolve overflow by \textbf{node splitting}.
2-4 Tree operations

24Tree::insert(k)
1. \( v \leftarrow 24\text{Tree}::\text{search}(k) \) // leaf where \( k \) should be
2. Add \( k \) and an empty subtree in key-subtree-list of \( v \)
3. while \( v \) has 4 keys (overflow \( \leadsto \) 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 \( v \)
4. \( v' \leftarrow \) new node with keys \( k_1, k_2 \) and subtrees \( T_0, T_1, T_2 \)
5. \( v'' \leftarrow \) new node with key \( k_4 \) and subtrees \( T_3, T_4 \)
6. Replace \( \langle v \rangle \) by \( \langle v', k_3, v'' \rangle \) in key-subtree-list of \( p \)
7. \( v \leftarrow p \)
Towards 2-4 Tree Deletion

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

![Diagram of a 2-4 tree with nodes 5, 9, 12, 3, 4, 6, 8, 11, 13, 14, 16]

- **Observe:** Any node except the root has an immediate sibling.
2-4 Tree Deletion

Example: \textit{delete}(43)

- \texttt{24Tree::search}, then trade with successor if KVP is not at a leaf.
2-4 Tree Deletion

**Example:** `delete(43)`

- `24Tree::search`, then trade with successor if KVP is not at a leaf.
- If underflow:
2-4 Tree Deletion

Example: delete(43)

- 24Tree::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: $\textit{delete}(19)$
- $\textit{24Tree::search}$, then trade with successor if KVP is not at a leaf.
- If underflow:
  - If immediate sibling has extras, $\textit{rotate/transfer}$
2-4 Tree Deletion

Example: \textit{delete}(19)

- \texttt{24Tree::search}, then trade with successor if KVP is not at a leaf.
- If underflow:
  - If immediate sibling has extras, \texttt{rotate/transfer}
  - Else \texttt{node merge} (this affects the parent!)
2-4 Tree Deletion

**Example:** delete(19)

- *24Tree::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)`

- **24Tree::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)

- 24Tree::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: \textit{delete}(42)

- \textit{24Tree::search}, then trade with successor if KVP is not at a leaf.
- If underflow:
  - If immediate sibling has extras, \textit{rotate/transfer}
  - Else \textit{node merge} (this affects the parent!)
**2-4 Tree Deletion**

**Example:** delete(42)

- *24Tree::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)`

- **24Tree::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

\[24\text{Tree::delete}(k)\]

1. \( v \leftarrow 24\text{Tree::search}(k) \) // node containing \( k \)
2. \textbf{if} \( v \) is not leaf
3. \hspace{1em} \textbf{swap} \( k \) with its successor \( k' \) and \( v \) with leaf containing \( k' \)
4. delete \( k \) and one empty subtree in \( v \)
5. \textbf{while} \( v \) has 0 keys (underflow)
6. \hspace{1em} \textbf{if} parent \( p \) of \( v \) is NIL, delete \( v \) and \textbf{break}
7. \hspace{1em} \textbf{if} \( v \) has immediate sibling \( u \) with 2 or more keys (transfer/rotate)
8. \hspace{2em} \textbf{transfer} the key of \( u \) that is nearest to \( v \) to \( p \)
9. \hspace{2em} \textbf{transfer} the key of \( p \) between \( u \) and \( v \) to \( v \)
10. \hspace{2em} \textbf{transfer} the subtree of \( u \) that is nearest to \( v \) to \( v \)
   \hspace{1em} \textbf{break}
11. \textbf{else} (merge & repeat)
12. \hspace{1em} \( u \leftarrow \) immediate sibling of \( v \)
13. \hspace{1em} \textbf{transfer} the key of \( p \) between \( u \) and \( v \) to \( u \)
14. \hspace{1em} \textbf{transfer} the subtree of \( v \) to \( u \)
15. \hspace{1em} delete node \( v \) and set \( v \leftarrow p \)
Deletion from a 2-4 Tree

\texttt{24Tree::delete}(k)
1. \texttt{v} ← \texttt{24Tree::search}(k) // node containing \texttt{k}
2. \texttt{if} \ \texttt{v} is not leaf
3. \hspace{1em} swap \texttt{k} with its successor \texttt{k}' and \texttt{v} with leaf containing \texttt{k}'
4. \texttt{delete \texttt{k} and one empty subtree in \texttt{v}}
5. \texttt{while} \ \texttt{v} has 0 keys (\texttt{underflow})
6. \hspace{1em} \texttt{if} parent \texttt{p} of \texttt{v} is NIL, delete \texttt{v} and \texttt{break}
7. \hspace{1em} \texttt{if} \ \texttt{v} has immediate sibling \texttt{u} with 2 or more keys (\texttt{transfer/rotate})
8. \hspace{2em} transfer the key of \texttt{u} that is nearest to \texttt{v} to \texttt{p}
9. \hspace{2em} transfer the key of \texttt{p} between \texttt{u} and \texttt{v} to \texttt{v}
10. \hspace{2em} transfer the subtree of \texttt{u} that is nearest to \texttt{v} to \texttt{v}
11. \hspace{1em} \texttt{break}
12. \texttt{else} (\texttt{merge & repeat})
13. \hspace{2em} \texttt{u} ← immediate sibling of \texttt{v}
14. \hspace{3em} transfer the key of \texttt{p} between \texttt{u} and \texttt{v} to \texttt{u}
15. \hspace{3em} transfer the subtree of \texttt{v} to \texttt{u}
16. \hspace{2em} delete node \texttt{v} and set \texttt{v} ← \texttt{p}
2-4 Tree summary

- A 2-4 tree has height $O(\log n)$
  - In internal memory, all operations have run-time $O(\log n)$.
  - This is no better than AVL-trees in theory.
    (Though 2-4-trees are faster than AVL-trees in practice, especially when converted to binary search trees called red-black trees. No details.)

- A 2-4 tree has height $\Omega(\log n)$
  - Level $i$ contains at most $4^i$ nodes
  - Each node contains at most 3 KVPs

- So not significantly better than AVL-trees w.r.t. block transfers.

- But we can generalize the concept to decrease the height.

  total: at most $3 \left( 1 + 4 + 4^2 + \ldots + 4^n \right) = 3 \cdot \frac{4^{n+1} - 1}{4 - 1}$
  - $n \leq 4^{h+1} \rightarrow \log n \leq 2(h+1)$.
Outline

1. External Memory
   - Motivation
   - Stream-based algorithms
   - External sorting
   - External Dictionaries
   - 2-4 Trees
   - \(a-b\)-Trees
   - B-Trees
**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 $d$ subtrees, then it stores $d-1$ 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:** $a \leq \lceil b/2 \rceil = \lceil (b+1)/2 \rceil$.

*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

\[
\begin{align*}
    b &= 6 \\
    a &= 3
\end{align*}
\implies 2 \leq \text{#keys per node} \leq 5
\]

A 3-6-tree

```
38
```
```
14 20 26 32
```
```
10 12 16 18 22 24 28 30
```
```
44 50 62
```
```
40 42 46 48 52 54 56 58 60 64 66
```
```
**a-b-tree insertion**

**insert**(55):

- Overflow now means $b$ keys (and $b + 1$ subtrees)
"a-b-tree insertion"

**insert(55):**

- Overflow now means \( b \) keys (and \( b + 1 \) subtrees)
- Node split \( \Rightarrow \) new nodes have \( \geq \lfloor (b-1)/2 \rfloor \) keys
- Since we required \( a \leq \lfloor (b+1)/2 \rfloor \), this is \( \geq a-1 \) keys as required.
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-$h$?

<table>
<thead>
<tr>
<th>Level</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\geq 1$</td>
</tr>
<tr>
<td>1</td>
<td>$\geq 2$</td>
</tr>
<tr>
<td>2</td>
<td>$\geq 2a$</td>
</tr>
<tr>
<td>3</td>
<td>$\geq 2a^2$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$h$</td>
<td>$\geq 2a^{h-1}$</td>
</tr>
</tbody>
</table>

\[
\# \text{nodes} \geq 1 + \sum_{i=0}^{h-1} 2a^i = 1 + (a-1)\sum_{i=0}^{h-1} 2a^i = 1 + 2(a-1)\frac{a^h}{a-1} = 1 + 2a^h
\]

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 $\Theta(\text{height})$ nodes
- Height is $O(\log n / \log a)$.
- Recall: $a \leq \lceil b/2 \rceil$ 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.

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

This is still no better than AVL-trees.

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

1. External Memory
   - Motivation
   - Stream-based algorithms
   - External sorting
   - External Dictionaries
   - 2-4 Trees
   - $a$-$b$-Trees
   - 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 in external memory

Close-up on one node in one block:

In this example: 17 computer-words fit into one block, so the $B$-tree can have order 6.
B-tree analysis

- search, insert, and delete each requires visiting $\Theta(\text{height})$ nodes
- Work within a node is done in internal memory $\implies$ 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.