B-trees
Motivation

- When the size of a data-structure is very big, it is kept on secondary memory (hard-disk/SSD).
- Secondary memory is slow compared to RAM.
- The time for reading a block in secondary memory consists of seek time (the delay between the request and the transfer of the data) and transfer time.
- The seek time is \(~ 10\) milliseconds in hard-disks, and \(~ 0.1\) milliseconds in SSDs.
- We want a data-structure that minimizes the disk accesses.
Motivation

- Suppose we need to store a dynamic set with $10^9$ elements on secondary memory.
- The height of a binary search tree is at least $\lfloor \log 10^9 \rfloor = 29$. Thus, a search operation in an optimal binary search tree requires 30 disk accesses in the worst case.

B-trees
A **B-tree** is a generalization of binary search tree, that can store many elements in one node.

If for example we store 1000 elements in a node (and each internal node has 1001 children), a tree of height 2 suffices. Thus, a search operation in an optimal B-tree requires 3 disk accesses in the worst case.

In fact, we can store the top two levels in RAM, and then a search operation requires only 1 disk access.
A B-tree is a rooted tree with the following properties.

1. A node $x$ has the following fields.
   1. $x.n$, the number of keys stored at $x$.
   2. The keys of the elements, $x.key_1, x.key_2, \ldots, x.key_{x.n}$ in increasing order ($x.key_1 \leq x.key_2 \leq \cdots$).
   3. Pointers to the satellite data of the keys.
   4. A boolean value $x.leaf$ which is TRUE if and only if $x$ is a leaf.

\[
\begin{array}{c}
14 \\
3 \quad 7 \quad 11 \\
1 \quad 2 \quad 4 \quad 5 \quad 6 \quad 8 \quad 9 \quad 10 \quad 12 \quad 13 \quad 15 \quad 16 \quad 17 \quad 19 \quad 20 \quad 22 \quad 23
\end{array}
\]
An internal node $x$ has $x.n + 1$ children. The node $x$ stores pointers to it children $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$.

The keys $x.key_1, \ldots, x.key_{x.n}$ separate the ranges of keys in the subtrees of $x$:

If $k_i$ is some key in the subtree of $x.c_i$ then

$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \cdots \leq x.key_{x.n} \leq k_{x.n+1}$.
4 All leaves have the same depth.
5 The tree has a fixed parameter $t$, called minimum degree, that defines upper and lower bounds on the number of keys stored at each node:
   1. Every node except the root must have at least $t - 1$ keys.
   2. If the tree is not empty, the root must have at least one key.
   3. Every node can have at most $2t - 1$ keys.
**The height of a B-tree**

**Theorem**

For $n \geq 1$, the height of a B-tree with minimum degree $t$ containing $n$ keys is at most $\log_t \frac{n+1}{2}$.

- Consider a B-tree of height $h$ with minimal number of keys: 1 key at the root and $t - 1$ keys at each other node.
- Number of keys in nodes of depth $i \geq 1$ is $2t^{i-1}(t - 1)$.

<table>
<thead>
<tr>
<th>depth</th>
<th>nodes</th>
<th>keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2(t-1)</td>
</tr>
<tr>
<td>2</td>
<td>2t</td>
<td>2t(t-1)</td>
</tr>
<tr>
<td>3</td>
<td>2t^2</td>
<td>2t^2(t-1)</td>
</tr>
</tbody>
</table>
The height of a B-tree

- The number of keys in a minimal B-tree is

\[ 1 + \sum_{i=1}^{h} 2t^{i-1}(t - 1) = 1 + 2(t - 1) \sum_{i=1}^{h} t^{i-1} = 1 + 2(t - 1) \left( \frac{t^h - 1}{t - 1} \right) = 2t^h - 1. \]

- For any B-tree we have \( n \geq 2t^h - 1. \)

- Therefore, \( h \leq \log_t \frac{n+1}{2}. \)
**Search**

**Search**\((x, k)\)

1. \(i \leftarrow 1\)
2. while \(i \leq x.n AND k > x.key_i\)
3. \(i \leftarrow i + 1\)
4. if \(i \leq x.n AND k = x.key_i\)
5. return \((x, i)\)
6. else if \(x.leaf\)
7. return NULL
8. else
9. Disk-Read\((x.c_i)\)
10. return **Search**\((x.c_i, k)\)

- We assume that the root is stored in RAM.
- CPU time: \(\Theta(th) = \Theta(t \log_t n)\).
- Disk accesses: \(h \leq \log_t n\).
**Searching**

\( \text{SEARCH}(x, k) \)

1. \( i \leftarrow 1 \)
2. \( \text{while } i \leq x.n \text{ AND } k > x.\text{key}_i \)
3. \( i \leftarrow i + 1 \)
4. \( \text{if } i \leq x.n \text{ AND } k = x.\text{key}_i \)
5. \( \text{return } (x, i) \)
6. \( \text{else if } x.\text{leaf} \)
7. \( \text{return } \text{NULL} \)
8. \( \text{else} \)
9. \( \text{DISK-READ}(x.c_i) \)
10. \( \text{return } \text{SEARCH}(x.c_i, k) \)

- We assume that the root is stored in RAM.
- CPU time: \( \Theta(th) = \Theta(t \log_t n) \).
- Disk accesses: \( h \leq \log_t n \).
Searching

**SEARCH**(\(x, k\))

1. \(i \leftarrow 1\)
2. while \(i \leq x.n\) AND \(k > x.key_i\)
3. \(i \leftarrow i + 1\)
4. if \(i \leq x.n\) AND \(k = x.key_i\)
5. return \((x, i)\)
6. else if \(x.leaf\)
7. return NULL
8. else
9. \(\text{DISK-READ}(x.c_i)\)
10. return \(\text{SEARCH}(x.c_i, k)\)

- We assume that the root is stored in RAM.
- CPU time: \(\Theta(th) = \Theta(t \log_t n)\).
- Disk accesses: \(h \leq \log_t n\).
**Searching**

\[ \text{Search}(x, k) \]

1. \( i \leftarrow 1 \)
2. \( \text{while } i \leq x.n \text{ AND } k > x.key_i \)
3. \( i \leftarrow i + 1 \)
4. \( \text{if } i \leq x.n \text{ AND } k = x.key_i \)
5. \( \text{return } (x, i) \)
6. \( \text{else if } x.\text{leaf} \)
7. \( \text{return } \text{NULL} \)
8. \( \text{else} \)
9. \( \text{Disk-Read}(x.c_i) \)
10. \( \text{return } \text{Search}(x.c_i, k) \)

- We assume that the root is stored in RAM.
- CPU time: \( \Theta(th) = \Theta(t \log_t n) \).
- Disk accesses: \( h \leq \log_t n \).
**Searching**

\( \text{SEARCH}(x, k) \)

1. \( i \leftarrow 1 \)
2. while \( i \leq x.n \) AND \( k > x.key_i \)
3. \( i \leftarrow i + 1 \)
4. if \( i \leq x.n \) AND \( k = x.key_i \)
5. return \((x, i)\)
6. else if \( x.leaf \)
7. return NULL
8. else
9. \( \text{DISK-READ}(x.c_i) \)
10. return \( \text{SEARCH}(x.c_i, k) \)

- We assume that the root is stored in RAM.
- CPU time: \( \Theta(th) = \Theta(t \log_t n) \).
- Disk accesses: \( h \leq \log_t n \).
Creating an empty B-tree

**CREATE**($T$)

1. $x \leftarrow \text{ALLOCATE-NODE}()$
2. $x.\text{leaf} \leftarrow \text{TRUE}$
3. $x.n \leftarrow 0$
4. $\text{DISKWRITE}(x)$
5. $T.\text{root} \leftarrow x$

- CPU time: $\Theta(1)$.
- Disk accesses: 1.
The goal is to insert the new element into an existing leaf.
This is easy if the leaf has less than $2t - 1$ elements.

Insert(T,15)
15 can be inserted to this leaf
The hard case is when the leaf has $2t - 1$ elements.

The solution is to perform the following step during the descend in the tree: Before entering a node, if the node has $2t - 1$ elements, perform node splitting on the node in order to reduce the number of elements.
Let \( v \) be a node with \( 2t - 1 \) elements. The node splitting operation on \( v \) replaces \( v \) with two nodes: one containing the \( t - 1 \) smallest elements of \( v \), and another containing the \( t - 1 \) largest elements.

The median element of \( v \) is moved to the parent of \( v \). If \( v \) is the root, a new root is created.

\[ t=3 \]

\[
\begin{array}{c}
10 & 40 & 60 \\
A & B & C \\
15 & 20 & 25 & 30 & 35 \\
D & E & F \\
& & & & & \\
\end{array}
\]

\[
\begin{array}{c}
10 & 25 & 40 & 60 \\
A & B & C \\
15 & 20 & 30 & 35 \\
D & E & F \\
& & & & & \\
\end{array}
\]
If $v$ is not the root, the parent of $v$ has less than $2t - 1$ elements, so it is legal to move the median to the parent.
Example

```
10 30 50
2 4 8
t=2
12 20 30 60 65
Insert(T,9)
```

B-trees
Example

Split node

$t=2$

Insert(T,9)

B-trees
Example

B-trees
Example

B-trees
Example

B-trees

Insert(T,9)

2 8 12 20 30 60 65
Example

9 is added to the leaf

B-trees
**SplitChild**($x, i$)

1. $y \leftarrow x.c_i$
2. $z \leftarrow \text{ALLOCATENode}()$
3. $z.\text{leaf} \leftarrow y.\text{leaf}$
4. $z.n \leftarrow t - 1$
5. for $j = 1$ to $t - 1$
6. \hspace{1em} $z.\text{key}_j \leftarrow y.\text{key}_{j+t}$
7. if not $y.\text{leaf}$
8. for $j = 1$ to $t$
9. \hspace{1em} $z.c_j \leftarrow y.c_{j+t}$
10. for $j = x.n + 1$ downto $i + 1$
11. \hspace{1em} $x.c_{j+1} \leftarrow x.c_j$
12. \hspace{1em} $x.c_{i+1} \leftarrow z$
13. for $j = x.n$ downto $i$
14. \hspace{1em} $x.\text{key}_{j+1} \leftarrow x.\text{key}_j$
15. \hspace{1em} $x.\text{key}_i \leftarrow y.\text{key}_t$
16. \hspace{1em} $x.n \leftarrow x.n + 1$
17. \hspace{1em} $y.n \leftarrow t - 1$
18. \hspace{1em} $\text{DISKWRITE}(y)$
19. \hspace{1em} $\text{DISKWRITE}(z)$
20. \hspace{1em} $\text{DISKWRITE}(x)$

**Figure:**

- $x$ with $i=2$
- $y$ with $n=5$ and leaf: FALSE
- $t=3$
**SplitChild**($x, i$)

1. $y \leftarrow x.c_i$
2. $z \leftarrow \text{ALLOCATENODE}()$
3. $z.\text{leaf} \leftarrow y.\text{leaf}$
4. $z.n \leftarrow t - 1$
5. for $j = 1$ to $t - 1$
   6. $z.\text{key}_j \leftarrow y.\text{key}_{j+t}$
   7. if not $y.\text{leaf}$
   8. for $j = 1$ to $t$
   9. $z.c_j \leftarrow y.c_{j+t}$
10. for $j = x.n + 1$ downto $i + 1$
11. $x.c_{j+1} \leftarrow x.c_j$
12. $x.c_{i+1} \leftarrow z$
13. for $j = x.n$ downto $i$
14. $x.\text{key}_{j+1} \leftarrow x.\text{key}_j$
15. $x.\text{key}_i \leftarrow y.\text{key}_t$
16. $x.n \leftarrow x.n + 1$
17. $y.n \leftarrow t - 1$
18. \text{DISKWRITE}(y)
19. \text{DISKWRITE}(z)
20. \text{DISKWRITE}(x)
**SplitChild**(\(x, i\))

1. \(y \leftarrow x.c_i\)
2. \(z \leftarrow \text{ALLOCATENODE}()\)
3. \(z.\text{leaf} \leftarrow y.\text{leaf}\)
4. \(z.n \leftarrow t - 1\)
5. **for** \(j = 1\) **to** \(t - 1\)
   6. \(z.\text{key}_j \leftarrow y.\text{key}_{j+t}\)
6. **if** not \(y.\text{leaf}\)
5. **for** \(j = 1\) **to** \(t\)
4. \(z.c_j \leftarrow y.c_{j+t}\)

(10) **for** \(j = x.n + 1\) **downto** \(i + 1\)
(11) \(x.c_{j+1} \leftarrow x.c_j\)
(12) \(x.c_{i+1} \leftarrow z\)
(13) **for** \(j = x.n\) **downto** \(i\)
(14) \(x.\text{key}_{j+1} \leftarrow x.\text{key}_j\)
(15) \(x.\text{key}_i \leftarrow y.\text{key}_t\)
(16) \(x.n \leftarrow x.n + 1\)
(17) \(y.n \leftarrow t - 1\)
(18) \(\text{DiskWrite}(y)\)
(19) \(\text{DiskWrite}(z)\)
(20) \(\text{DiskWrite}(x)\)
SplitChild($x$, $i$)

1. $y \leftarrow x.c_i$
2. $z \leftarrow \text{ALLOCATENode}()$
3. $z.\text{leaf} \leftarrow y.\text{leaf}$
4. $z.n \leftarrow t - 1$
5. \textbf{for} $j = 1$ \textbf{to} $t - 1$
6. \hspace{1em} $z.\text{key}_j \leftarrow y.\text{key}_{j+t}$
7. \hspace{1em} \textbf{if} not $y.\text{leaf}$
8. \hspace{2em} \textbf{for} $j = 1$ \textbf{to} $t$
9. \hspace{3em} $z.c_j \leftarrow y.c_{j+t}$
10. \textbf{for} $j = x.n + 1$ \textbf{downto} $i + 1$
11. \hspace{1em} $x.c_{j+1} \leftarrow x.c_j$
12. \hspace{1em} $x.c_{i+1} \leftarrow z$
13. \hspace{1em} \textbf{for} $j = x.n$ \textbf{downto} $i$
14. \hspace{2em} $x.\text{key}_{j+1} \leftarrow x.\text{key}_j$
15. \hspace{2em} $x.\text{key}_i \leftarrow y.\text{key}_t$
16. \hspace{1em} $x.n \leftarrow x.n + 1$
17. \hspace{1em} $y.n \leftarrow t - 1$
18. \hspace{1em} \text{DISKWRITE}($y$)
19. \hspace{1em} \text{DISKWRITE}($z$)
20. \hspace{1em} \text{DISKWRITE}($x$)

---

**B-trees**
**SplitChild**$(x, i)$

1. $y \leftarrow x.c_i$
2. $z \leftarrow $ ALLOCATENODE$(\cdot)$
3. $z.\text{leaf} \leftarrow y.\text{leaf}$
4. $z.n \leftarrow t - 1$
5. for $j = 1$ to $t - 1$
   6. $z.\text{key}_j \leftarrow y.\text{key}_{j+t}$
   7. if not $y.\text{leaf}$
   8. for $j = 1$ to $t$
   9. $z.c_j \leftarrow y.c_{j+t}$
10. for $j = x.n + 1$ downto $i + 1$
11. $x.c_{j+1} \leftarrow x.c_j$
12. $x.c_{i+1} \leftarrow z$
13. for $j = x.n$ downto $i$
14. $x.\text{key}_{j+1} \leftarrow x.\text{key}_j$
15. $x.\text{key}_i \leftarrow y.\text{key}_t$
16. $x.n \leftarrow x.n + 1$
17. $y.n \leftarrow t - 1$
18. DISKWRITE$(y)$
19. DISKWRITE$(z)$
20. DISKWRITE$(x)$

---

**B-trees**

$B \leftarrow C \leftarrow A$

$t = 3$

<table>
<thead>
<tr>
<th>Key</th>
<th>n:</th>
<th>Leaf</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>FALSE</td>
</tr>
<tr>
<td>40</td>
<td>3</td>
<td>FALSE</td>
</tr>
<tr>
<td>60</td>
<td>3</td>
<td>FALSE</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>FALSE</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>FALSE</td>
</tr>
<tr>
<td>25</td>
<td>5</td>
<td>FALSE</td>
</tr>
<tr>
<td>30</td>
<td>5</td>
<td>FALSE</td>
</tr>
<tr>
<td>35</td>
<td>5</td>
<td>FALSE</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>FALSE</td>
</tr>
<tr>
<td>35</td>
<td>2</td>
<td>FALSE</td>
</tr>
<tr>
<td>40</td>
<td>3</td>
<td>FALSE</td>
</tr>
<tr>
<td>60</td>
<td>3</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

- $A \leftarrow B \leftarrow C \leftarrow D \leftarrow E \leftarrow F$
- $D \leftarrow E \leftarrow F$
**SplitChild**($x, i$)

1. $y \leftarrow x.c_i$
2. $z \leftarrow \text{ALLOCATENode}()$
3. $z.\text{leaf} \leftarrow y.\text{leaf}$
4. $z.n \leftarrow t - 1$
5. **for** $j = 1$ **to** $t - 1$
6. \hspace{1em} $z.\text{key}_j \leftarrow y.\text{key}_{j+t}$
7. **if** not $y.\text{leaf}$
8. \hspace{1em} **for** $j = 1$ **to** $t$
9. \hspace{2em} $z.c_j \leftarrow y.c_{j+t}$

10. **for** $j = x.n + 1$ **downto** $i + 1$
11. \hspace{1em} $x.c_{j+1} \leftarrow x.c_j$
12. \hspace{1em} $x.c_{i+1} \leftarrow z$
13. **for** $j = x.n$ **downto** $i$
14. \hspace{1em} $x.\text{key}_{j+1} \leftarrow x.\text{key}_j$
15. \hspace{1em} $x.\text{key}_i \leftarrow y.\text{key}_t$
16. \hspace{1em} $x.n \leftarrow x.n + 1$
17. \hspace{1em} $y.n \leftarrow t - 1$
18. **DISKWRITE**(y)
19. **DISKWRITE**(z)
20. **DISKWRITE**(x)
**SplitChild**($x, i$)

1. $y \leftarrow x.c_i$
2. $z \leftarrow \text{ALLOCATENODE}()$
3. $z.\text{leaf} \leftarrow y.\text{leaf}$
4. $z.n \leftarrow t - 1$
5. for $j = 1$ to $t - 1$
   6. $z.\text{key}_j \leftarrow y.\text{key}_{j+t}$
   7. if not $y.\text{leaf}$
      8. for $j = 1$ to $t$
         9. $z.c_j \leftarrow y.c_{j+t}$
10. for $j = x.n + 1$ downto $i + 1$
11. $x.c_{j+1} \leftarrow x.c_j$
12. $x.c_{i+1} \leftarrow z$
13. for $j = x.n$ downto $i$
14. $x.\text{key}_{j+1} \leftarrow x.\text{key}_j$
15. $x.\text{key}_i \leftarrow y.\text{key}_t$
16. $x.n \leftarrow x.n + 1$
17. $y.n \leftarrow t - 1$
18. DiskWrite($y$)
19. DiskWrite($z$)
20. DiskWrite($x$)

---

$B$-trees
SplitChild($x$, $i$)

1. $y \leftarrow x.c_i$
2. $z \leftarrow \text{ALLOCATENode}()$
3. $z.\text{leaf} \leftarrow y.\text{leaf}$
4. $z.n \leftarrow t - 1$
5. for $j = 1$ to $t - 1$
6. $z.\text{key}_j \leftarrow y.\text{key}_{j+t}$
7. if not $y.\text{leaf}$
8. for $j = 1$ to $t$
9. $z.c_j \leftarrow y.c_{j+t}$
10. for $j = x.n + 1$ downto $i + 1$
11. $x.c_{j+1} \leftarrow x.c_j$
12. $x.c_{i+1} \leftarrow z$
13. for $j = x.n$ downto $i$
14. $x.\text{key}_{j+1} \leftarrow x.\text{key}_j$
15. $x.\text{key}_i \leftarrow y.\text{key}_t$
16. $x.n \leftarrow x.n + 1$
17. $y.n \leftarrow t - 1$
18. DiskWrite($y$)
19. DiskWrite($z$)
20. DiskWrite($x$)

B-trees
**Insert — Pseudo-code**

\[
\text{\textbf{Insert}}(T, k)
\]

1. \( r \leftarrow T.\text{Root} \)
2. \textbf{if} \( r.n = 2t - 1 \)
3. \( s \leftarrow \text{AllocateNode}() \)
4. \( T.\text{root} \leftarrow s \)
5. \( s.\text{leaf} \leftarrow \text{FALSE} \)
6. \( s.n \leftarrow 0 \)
7. \( s.c_1 \leftarrow r \)
8. \( \text{SplitChild}(s, 1) \)
9. \( \text{InsertNonfull}(s, k) \)
10. \textbf{else}
11. \( \text{InsertNonfull}(r, k) \)
**Insert — Pseudo-code**

\[
\text{INSERTNONFULL}(x, k) \\
\text{(1)} \quad i \gets x.n \\
\text{(2)} \quad \text{if } x.\text{leaf} \\
\text{(3)} \quad \text{while } i \geq 1 \text{ AND } k < x.\text{key}_i \\
\text{(4)} \quad x.\text{key}_{i+1} \gets x.\text{key}_i \\
\text{(5)} \quad i \gets i - 1 \\
\text{(6)} \quad x.\text{key}_{i+1} \gets k \\
\text{(7)} \quad x.n \gets x.n + 1 \\
\text{(8)} \quad \text{DISKWRITE}(x) \\
\text{(9)} \quad \text{else} \\
\text{(10)} \quad \text{while } i \geq 1 \text{ AND } k < x.\text{key}_i \\
\text{(11)} \quad i \gets i - 1 \\
\text{(12)} \quad i \gets i + 1 \\
\text{(13)} \quad \text{DISKREAD}(x.c_i) \\
\text{(14)} \quad \text{if } x.c_i.n = 2t - 1 \\
\text{(15)} \quad \text{SPLITCHILD}(x, i) \\
\text{(16)} \quad \text{if } k > x.\text{key}_i \\
\text{(17)} \quad i \gets i + 1 \\
\text{INSERTNONFULL}(x.c_i, k) \]

B-trees
The easy case is deleting an element from a leaf that has more than $t - 1$ elements.

The hard cases are deleting an item from a leaf that has $t - 1$ elements, and deleting an item from an internal node.

The following step is performed during the descend in the tree: Before entering a node $v$, if $v$ has $t - 1$ elements, perform shifting or merging in order to increase the number of elements.
If \( v \) has an immediate left sibling \( u \) and \( u \) has at least \( t \) elements:

1. Move the element in the parent of \( v \) that “separates” \( v \) and \( u \) to \( v \).
2. Move the maximum element of \( u \) to the parent of \( v \).

\[
\begin{array}{c|c|c|c|c}
10 & 40 & & & \\
20 & 25 & 30 & E & F \\
A & B & C & D & E & F \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
10 & 30 & & & \\
20 & 25 & 40 & 50 & \\
A & B & C & D & E & F \\
\end{array}
\]
Otherwise, if \( v \) has an immediate right sibling \( w \), and \( w \) has at least \( t \) elements then

1. Move the element of the parent of \( v \) that “separates” \( v \) and \( w \) to \( v \).
2. Move the minimum element of \( w \) to the parent of \( v \).

\[
\begin{array}{cc}
10 & 40 \\
5 & 30 & 50 & 60 \\
A & B & C & D & E \\
\end{array}
\quad t=2
\begin{array}{cc}
10 & 50 \\
5 & 30 & 40 & 60 \\
A & B & C & D & E \\
\end{array}
\]
If $v$ has no immediate sibling with at least $t$ elements:

1. Merge $v$ with an immediate sibling (merge with the immediate left sibling if it exists, and otherwise merge with the right immediate right sibling).
2. Move the “separating element” from the parent of $v$ to the new node.

```
10 40
A  B
  30  50
    v
10

A  B  C  D
30 40 50
```

$t=2$
When reaching the node $x$ that contains the key $k$ we want to delete, there are several cases.

**Case 1:** If $x$ is a leaf, delete the key.
Deleting a key — Case 2

Suppose that $x$ is an internal node, and $k = x.key_i$. Let $y = x.c_i$ and $z = x.c_{i+1}$.

**Case 2:** If $y$ has at least $t$ elements, recursively delete the predecessor of $k$ (which is in the subtree of $y$) and put it in $x$ instead of $k$. 

```
Delete(x,40)  
10 40  
20 30
23 26 35
```

$t=2$
Deleting a key — Case 2

Suppose that $x$ is an internal node, and $k = x.key_i$. Let $y = x.c_i$ and $z = x.c_{i+1}$.

**Case 2:** If $y$ has at least $t$ elements, recursively delete the predecessor of $k$ (which is in the subtree of $y$) and put it in $x$ instead of $k$. 

![Diagram](image)
Deleting a key — Case 2

Suppose that $x$ is an internal node, and $k = x.key_i$. Let $y = x.c_i$ and $z = x.c_{i+1}$.

**Case 2**: If $y$ has at least $t$ elements, recursively delete the predecessor of $k$ (which is in the subtree of $y$) and put it in $x$ instead of $k$. 

\[
\begin{array}{c}
x & & \text{Delete(x,40)} \\
& 10 & 35 & \\
y & & z \\
& 20 & 26 & \\
& & 23 & 30 \\
\end{array}
\]
Case 3: If $y$ has $t - 1$ elements and $z$ has at least $t$ elements, recursively delete the successor of $k$ (which is in the subtree of $z$) and put it in $x$ instead of $k$. 

$$\text{Delete}(x, 40)$$
Case 3: If $y$ has $t - 1$ elements and $z$ has at least $t$ elements, recursively delete the successor of $k$ (which is in the subtree of $z$) and put it in $x$ instead of $k$. 

```
Delete(x, 40)
```

```
10 43
```

```
30
```

```
50 60
```

```
46
```
Case 4: If both $y$ and $z$ have $t - 1$ elements, merge $y$ and $z$, and recursively delete $k$ from the merged node.

![B-tree diagram]

Delete(x, 40)

$\begin{array}{c}
\text{Delete(x, 40)} \\
t=2
\end{array}$
Case 4: If both $y$ and $z$ have $t - 1$ elements, merge $y$ and $z$, and recursively delete $k$ from the merged node.
Case 4: If both \( y \) and \( z \) have \( t - 1 \) elements, merge \( y \) and \( z \), and recursively delete \( k \) from the merged node.

Delete(\( x, 40 \))

\[ t = 2 \]
Case 4: If both $y$ and $z$ have $t-1$ elements, merge $y$ and $z$, and recursively delete $k$ from the merged node.
Example

\[ t=3 \]
\[ \text{delete}(T,6) \]

B-trees
6 < 14 so the next node is the 1st child of the current node. This child has $3 > t - 1$ elements, so no modification is needed.
Example

3 < 6 < 7 so the next node is the 2nd child of the current node. This child has $3 > t - 1$ elements, so no modification is needed.
Example

```
t=3
18 21
8 9 10 15 16 17 19 20 22 23
4 5 6 1 2
3 7 11
14
12 13
```

```
delete(T, 6)
```
Example

The element 6 is deleted from the current node.
Example

B-trees
Example

\[ t=3 \]
\[ \text{delete}(T, 11) \]

11 < 14 so the next node is the 1st child of the current node. This child has 3 > \( t - 1 \) elements, so no modification is needed.
11 is the 3rd item of the node. The 4th child has $2 = t - 1$ elements, so we don’t replace 11 with its successor.

The 3rd child has $3 > t - 1$ so we replace 11 by its predecessor and delete the predecessor.
Example

t=3

delete(T, 11)

B-trees
The element 10 is deleted from the current node.
The element 11 is replaced by 10.
Example

t=3
delete(T, 7)

B-trees
delete(T, 7)

7 < 14 so the next node is the 1st child of the current node. This child has 3 > t − 1 elements, so no modification is needed.
Example

$t=3$

18 21
8 9 15 16 17 19 20 22 23
12 13
4 5 1 2
3 7 10
14

delete(T, 7)

7 is the 2nd item of the node.
Both the 2nd and 3rd children of the node have \( 2 = t - 1 \) elements. Therefore we merge these children and move 7 to the new node.
Example

t=3

delete(T, 7)

B-trees
Example

The element 7 is deleted from the current node.
Example

B-trees
Example

4 < 14 so the next node is the 1st child of the current node. This child has 2 > t − 1 elements, so a modification is needed. This child has no immediate sibling with more than 2 = t − 1 elements, so perform merging on this child and its sibling. The height of the tree decreases by 1.
Example

3 < 4 < 10 so the next node is the 2nd child of the current node.
This child has $4 > t - 1$ elements, so no modification is needed.
Example

15 16 17 19 20 22 23
1 2 3 10 14 18 21

\text{delete}(T, 4)

B-trees
The element 4 is deleted from the current node.
Example

t=3
delete(T,2)

B-trees
Example

t = 3

delete(T, 2)

2 < 3 so the next node is the 1st child of the current node.
This child has 2 = t – 1 elements, so a modification is needed.
The immediate sibling of this child has 3 > t – 1 elements, so
we perform shifting.
Example

t=3
delete(T, 2)

B-trees
The element 2 is deleted from the current node.