External Memory Dictionary

**Task:** Given a large amount of data that does not fit into main memory, process it into a dictionary data structure.

- Need to minimize number of disk accesses
- With each disk read, read a whole block of data
- Construct a balanced search tree that uses one disk block per tree node
- Each node needs to contain more than one key
From Binary to k-ary

A k-ary search tree $T$ is defined as follows:

For each node $x$ of $T$:

- $x$ has at most $k$ children

- $x$ stores an ordered list of pointers to its children

- $x$ stores an ordered list of keys

- $x$ fulfills the search tree property: keys in the subtree rooted at $i$-th child $\leq$ keys in subtree rooted at $(i + 1)$-st child.
Chapter 18: B-Trees

A B-tree is a balanced tree scheme in which balance is achieved by permitting the nodes to have multiple keys and more than two children.
**Definition** Let \( t \geq 2 \) be an integer. A tree \( T \) is called a **B-tree** having **minimum degree** \( t \) if the leaves of \( T \) are at the same depth and each node \( u \) has the following properties:

1. \( u \) has at most \( 2t - 1 \) keys.
2. If \( u \) is not the root, \( u \) has at least \( t - 1 \) keys.
3. The keys in \( u \) are sorted in the increasing order.
4. The number of \( u \)'s children is precisely one more than the number of \( u \)'s keys.
5. For all \( i \geq 1 \), if \( u \) has at least \( i \) keys and has children, then every key appearing in the subtree rooted at the \( i \)-th child of \( u \) is less than the \( i \)-th key and every key appearing in the subtree rooted at the \( (i + 1) \)-st child of \( u \) is greater than the \( i \)-th key.
Notation

Let $u$ be a node in a B-tree. By $n[u]$ we denote the number of keys in $u$. For each $i$, $1 \leq i \leq n[u]$, $key_i[u]$ denotes the $i$-th key of $u$. For each $i$, $1 \leq i \leq n[u] + 1$, $c_i[u]$ denotes the $i$-th child of $u$.

Terminology

For the sake of simplicity we will introduce some terminology.

We say that a node is full if it has $2t - 1$ keys and we say that a node is lean if it has the minimum number of keys, that is $t - 1$ keys in the case of a non-root and 1 in the case of the root.

2-3-4 Trees

B-trees with minimum degree 2 are called 2-3-4 trees to signify that the number of children is two, three, or four.
Depth of a B-tree

**Theorem A**  Let $t \geq 2$ and $n$ be integers. Let $T$ be an arbitrary B-tree with minimum degree $t$ having $n$ keys. Let $h$ be the height of $T$. Then $h \leq \log_t \frac{n+1}{2}$.

**Proof** The height is maximized when all the nodes are lean. If $T$ is of that form, the number of keys in $T$ is

$$1 + \sum_{i=1}^{h} 2(t-1)t^{i-1} = 2(t-1)\frac{t^h - 1}{t - 1} + 1 = 2t^h - 1.$$

Thus the depth of a B-tree is at most $\frac{1}{\log t}$ of the depth of an RB-tree.
Searching for a key $k$ in a B-tree

Start with $x = root[T]$.

1. If $x = \text{nil}$, then $k$ does not exist.
2. Compute the smallest $i$ such that the $i$-th key at $x$ is greater than or equal to $k$.
3. If the $i$-th key is equal to $k$, then the search is done.
4. Otherwise, set $x$ to the $i$-th child.

The number of examinations of the key is

$$O((2t - 1)h) = O(t \log_t (n + 1)/2).$$

Binary search may improve the search within a node

\[\text{How do we search for a predecessor?}\]
B-tree searching runtime

- $O(t)$ per node

- Path has height $h = O(\log_t n)$

- CPU-time: $O(t \log_t n)$

Disk accesses: $O(\log_t n)$. Disk accesses are more expensive than CPU time
B-Tree-Predecessor($T, x, i$)

1: \[\triangleright \text{Find a pred. of } key_i[x] \text{ in } T\]
2: \[\textbf{if } i \geq 2 \textbf{ then}\]
3: \[\textbf{if } c_i[x] = \text{nil} \textbf{ then return } key_{i-1}[x]\]
4: \[\triangleright \text{If } i \geq 2 & x \text{ is a leaf}\]
5: \[\triangleright \text{return the } (i-1)\text{st key}\]
6: \[\textbf{else}\{\]
7: \[\triangleright \text{If } i \geq 2 & x \text{ is not a leaf}\]
8: \[\triangleright \text{find the rightmost key in the } i\text{-th child}\]
9: \[y \leftarrow c_i[x]\]
10: \[\textbf{repeat}\]
11: \[z \leftarrow c_n[y]+1\]
12: \[\textbf{if } z \neq \text{nil} \textbf{ then } y \leftarrow z\]
13: \[\textbf{until } z = \text{nil}\]
14: \[\textbf{return } key_n[y][y]\]
15: \}
16: else {
17: \[\triangleright\text{Find } y \text{ and } j \geq 1 \text{ such that}\]
18: \[\triangleright\text{x is the leftmost key in } c_j[y]\]
19: \[\textbf{while } y \neq \text{root}[T] \text{ and } c_1[p[y]] = y \text{ do}\]
20: \[y \leftarrow p[y]\]
21: \[j \leftarrow 1\]
22: \[\textbf{while } c_j[p[y]] \neq y \text{ do } j \leftarrow j + 1\]
23: \[\textbf{if } j = 1 \text{ then return } \text{“No Predecessor”}\]
24: \[\textbf{return } key_{j-1}[p[y]]\]
25: }
Basic Operations, Split & Merge

Split takes a full node $x$ as part of the input. If $x$ is not the root, then its parent should not be full. The input node is split into three parts: the middle key, a node that has everything to the left of the middle key, and a node that has everything to the right of the middle key. Then the three parts replace the pointer pointing to $x$. As a result, in the parent node the number of children and the number of keys are both increased by one.
inserted

node y

node x

node z

\text{t=4}
B-Tree-Split($T, x$)
1:  \textbf{if} $n[x] < 2t - 1$ \textbf{then return} “Can’t Split”
2:  \textbf{if} $x \neq \text{root}[T]$ \textbf{and} $n[p[x]] = 2t - 1$ \textbf{then}
3:  \textbf{return} “Can’t Split”
4:  $\triangleright$ Create new nodes $y$ and $z$
5:  $n[y] \leftarrow t - 1$
6:  $n[z] \leftarrow t - 1$
7:  $\textbf{for } i \leftarrow 1 \textbf{ to } t - 1 \textbf{ do}$ \{ 
8:  $\text{key}_i[y] \leftarrow \text{key}_i[x]$
9:  $\text{key}_i[z] \leftarrow \text{key}_{i+t}[x]$
10:  \}
11:  $\textbf{for } i \leftarrow 1 \textbf{ to } t \textbf{ do}$ \{ 
12:  $c_i[y] \leftarrow c_i[x]$
13:  $c_i[z] \leftarrow c_{i+t}[x]$
14:  \}
15:  $\triangleright$ If $x$ is the root then create a new root
16:  $\textbf{if } x = \text{root}[T]$ \textbf{then}$ \{ 
17:  $\triangleright$ Create a new node $v$
18:  $n[v] \leftarrow 0$
19:  $c_1[v] \leftarrow x$
20:  $p[x] \leftarrow v$
21:  $\text{root}[T] \leftarrow v$
22:  \}
23: ▷ Find the spot for insertion
24: \( j \leftarrow 1 \)
25: \textbf{while} \( c_j[p[x]] \neq x \) \textbf{do} \( j \leftarrow j + 1 \)
26: ▷ Open up space for insertion
27: \textbf{if} \( j \leq n[p[x]] \) \textbf{then}
28: \textbf{for} \( i \leftarrow n[p[x]] \) \textbf{downto} \( j \) \textbf{do} \{ 
29: \hspace{1cm} \text{key}_{i+1}[p[x]] \leftarrow \text{key}_i[p[x]] \\
30: \hspace{1cm} c_{i+2}[p[x]] \leftarrow c_{i+1}[p[x]] \\
31: \} \\
32: ▷ Insertion
33: \text{key}_j[p[x]] \leftarrow \text{key}_t[x] \\
34: \text{c}_j[p[x]] \leftarrow y \\
35: \text{c}_{j+1}[p[x]] \leftarrow z \\
36: n[p[x]] \leftarrow n[p[x]] + 1 \\
37: p[y] \leftarrow p[x] \\
38: p[z] \leftarrow p[x] \\
39: ▷ Return the pointer to the parent \\
40: \textbf{return} p[x]
Merge takes as input a node and the position of a key. Then it merges the key and the pair of children flanking the key into one node.

What kind of properties must the input node and the children satisfy for such an operation be possible?
Well, the input node must not be lean, and the two children must be lean.

That’s right.
node x dropped

node y

node z

t=4
B-Tree-Merge($T, x, i$)

1: $\triangleright$ Merge the $i$-th key of $x$ and the two
2: $\triangleright$ children flanking the $i$-th key
3: $y \leftarrow c_i[x]$ $\triangleright y$ is the left child
4: $z \leftarrow c_{i+1}[x]$ $\triangleright z$ is the right child
5: if ($n[y] > t - 1$ or $n[z]] > t - 1$) then
6: return “Can’t Merge”
7: $key_t[y] \leftarrow key_i[x]$ $\triangleright$ Append the middle key
8: for $j \leftarrow 1$ to $t - 1$ do $\triangleright$ Copy keys from $z$
9: $key_{t+j}[y] \leftarrow key_j[z]$
10: for $j \leftarrow 1$ to $t$ do {
11: $c_{t+j}[y] \leftarrow c_j[z]$ $\triangleright$ Copy children from $z$
12: $p[c_j[z]] \leftarrow y$ $\triangleright$ Fix the parent pointers
13: }
14: $n[x] \leftarrow n[x] - 1$ $\triangleright$ Fix the $n$-tag
15: if ($n[x] = 0$) then {
16: $root[T] \leftarrow y$ $\triangleright$ and was lean, then
17: $p[y] \leftarrow nil$ $\triangleright y$ becomes the root
18: }
19: ▷ If the middle key is not the last key
20: ▷ Fill the gap by moving things
21:  else if $i \leq n[x]$ then { 
22:      for $j \leftarrow i$ to $n[x]$ do 
23:         $key_j[x] \leftarrow key_{j+1}[x]$
24:      for $j \leftarrow i$ to $n[x]$ do 
25:         $c_j[x] \leftarrow c_{j+1}[x]$
26:  }
**Insertion of a key**

Suppose that a key $k$ needs to be inserted in the subtree rooted at $y$ in a B-tree $T$.

As with a binary search tree, we can insert a key into a B-tree in a single pass down the tree from the root to a leaf. To do so, we do not wait to find out whether we will actually need to split a full node in order to do the insertion. Instead, as we travel down the tree searching for the position for the new key, we split each full node we come to along the way (including the leaf itself). Thus whenever we want to split a full node, we are assured that its parent is not full.

If $y$ is a leaf, insert the key. If not, find a child in which the key should go to and then make a recursive call with $y$ set to the child.
the initial tree

B inserted

Q inserted

L inserted

F inserted

P
Runtime of Insertion

- $O(t)$ per node
- Path has height $h = O(\log_t n)$
- CPU-time: $O(t \log_t n)$

Disk accesses: $O(\log_t n)$. Disk accesses are more expensive than CPU time
B-Tree-Insert\((T, y, k)\)

1: \(z \leftarrow y\)
2: \(f \leftarrow \text{false}\)
3: \(\text{while } f = \text{false} \text{ do } \{\)
4: \(\text{if } n[z] = 2t - 1 \text{ then}\)
5: \(z \leftarrow \text{B-Tree-Split}(T, z)\)
6: \(j \leftarrow 1\)
7: \(\text{while } \text{key}_j[z] < k \text{ and } j \leq n[z] \text{ do}\)
8: \(j \leftarrow j + 1\)
9: \(\text{if } c_j[z] \neq \text{nil} \text{ then } z \leftarrow c_j[z]\)
10: \(\text{else } f \leftarrow \text{true}\)
11: \}\)
12: \(\text{for } i \leftarrow n[z] \text{ downto } j \text{ do}\)
13: \(\text{key}_{j+1}[z] \leftarrow \text{key}_j[z]\)
14: \(\text{key}_j \leftarrow k\)
15: \(n[z] \leftarrow n[z] + 1\)
16: \(\text{return } z\)
Deletion

The task is to receive a key $k$ and a B-tree $T$ as input and eliminate it from $T$ if it is in the tree. To accomplish this, we will take an approach similar to that we took for binary search trees.

- Search for $k$. If the node containing $k$ is a leaf, eliminate $k$.
- Otherwise, search for the predecessor $k$. Relocate the predecessor to the position of $k$.

What should we be careful about?
Issues

Just as a simple insertion algorithm might have to back up if a node on the path to where the keys was to be inserted was full, a simple approach to deletion might have to back up if a node (other than the root) along the path to where the key to be deleted has the minimum number of keys.
Well, we should avoid removing a key from a lean leaf.

To avoid such a case, probably we should take a strategy similar to that we took in Insertion, that is, when a node is about to be visited, make sure that the node is not lean.

*That's correct.*
Strategy

When a lean node $x$ is about to be visited, do the following:

- In the case when $x$ is not the first child, if its immediate left sibling is not lean move the last key and the last child of the sibling to $x$; otherwise merge the sibling, $x$, and the key between them into one.

- In the case when $x$ is the first child, if its immediate right sibling is not lean move the first key and the first child of the sibling to $x$; otherwise merge the sibling, $x$, and the key between them into one.

We can then assume that if $x$ is not the root then its parent is not lean.
B-Tree-Delete($T, k$)

1: \(\triangleright\) Search for $k$
2: \(x \leftarrow \text{root}[T] \)
3: if \(n[x] = 1 \text{ and } n[c_1[x]] = t - 1 \text{ and } n[c_1[x]] = t - 1\) then B-Tree-Merge($T, x, 1$)
4: \(f \leftarrow \text{false} \)
5: while \(x \neq \text{nil} \text{ or } f = \text{false}\) do \{
6: \(q \leftarrow 1 \)
7: while \((q \leq n[x] \text{ and } \text{key}_q[x] \geq k)\) do
8: \(q \leftarrow q + 1\) \(\triangleright\) Scan the keys
9: if \((q \leq n[x] \text{ and } \text{key}_q[x] = k)\) then
10: \(f \leftarrow \text{true} \quad \triangleright k \text{ has been found} \)
11: else \{
12: \}


if \( c_q[x] \neq \text{nil} \) and \( n[c_q[x]] = t - 1 \) then

If \( c_q[x] \) exists and is lean, then do the following

if \( q \neq n[x] + 1 \) then

If \( c_q[x] \) has the immediate right sibling

if \( n[c_q+1[x]] = t - 1 \) then

If the sibling is lean, merge it with \( c_q[x] \)

B-Tree-Merge\((T, x, q)\)

Otherwise, ship a key and a child from it

else B-Tree-GetFromRight\((T, x, q)\)

else

If \( c_q[x] \) is the rightmost child

If the immediate left sibling is lean

if \( n[c_q-1[x]] = t - 1 \) then {

merge it with \( c_q[x] \) and fix \( q \)

\[ q \leftarrow q - 1 \]

B-Tree-Merge\((T, x, q)\) }

Otherwise, ship a key and a child from it

else B-Tree-GetFromLeft\((T, x, q)\)

\[ x \leftarrow c_q[x] \]

}
if $x = \text{nil}$ then return “Key Not Found”

if $c_q[x] = \text{nil}$ then {
    $n[x] \leftarrow n[x] - 1$
    for $i \leftarrow q$ to $n[x] - 1$ do
        $key_i[x] \leftarrow key_{i+1}[x]$
    return 
}

$y \leftarrow x$ \quad ▷ Now find the predecessor
$r \leftarrow q$

while $c_r[y] \neq \text{nil}$ do {
    if $n[c_r[y]] = t - 1$ then
        if $r \neq n[y] + 1$ then
            if $n[c_{r+1}[y]] = t - 1$ then
                B-Tree-Merge($T, y, r$)
            else B-Tree-GetFromRight($T, y, r$)
        else if $n[c_{r-1}[y]] = t - 1$ then {
            $r \leftarrow r - 1$
            B-Tree-Merge($T, y, r$) }
        else B-Tree-GetFromLeft($T, y, r$) }  
    $y \leftarrow c_r[y]$
    $r \leftarrow n[y] + 1$ }

$key_q[x] \leftarrow key_r[y]$ \quad ▷ Relocate the key
$n[y] \leftarrow n[y] - 1$
B-Tree-GetFromRight\((T, x, i)\)

1: \( y \leftarrow c_i[x] \)
2: \( z \leftarrow c_{i+1}[x] \)
3: \( n[y] \leftarrow n[y] + 1 \)
4: \( key_{n[y]}[y] \leftarrow key_i[x] \)
5: \( key_i[x] \leftarrow key_{1}[z] \)
6: \( c_{n[y]+1}[y] \leftarrow c_1[z] \)
7: \( p[c_1[x]] \leftarrow y \)
8: \( n[z] \leftarrow n[z] - 1 \)
9: for \( i \leftarrow 1 \) to \( n[z] \) do \( key_i[z] \leftarrow key_{i+1}[z] \)
10: for \( i \leftarrow 1 \) to \( n[z] + 1 \) do \( c_i[z] \leftarrow c_{i+1}[z] \)
B-Tree-GetFromLeft($T, x, i$)

1: $y \leftarrow c_{i}[x]$
2: $z \leftarrow c_{i-1}[x]$
3: for $i \leftarrow n[y]$ downto 1 do
   4: $key_{i+1}[z] \leftarrow key_{i}[z]$
5: for $i \leftarrow n[y] + 1$ downto 1 do
6: $c_{i+1}[z] \leftarrow c_{i}[z]$
7: $n[y] \leftarrow n[y] + 1$
8: $key_{1}[y] \leftarrow key_{i-1}[x]$
9: $key_{i-1}[x] \leftarrow key_{n[z]}[z]$
10: $c_{1}[y] \leftarrow c_{n[z]+1}[z]$
11: $p[c_{n[z]+1}[z]] \leftarrow y$
12: $n[z] \leftarrow n[z] - 1$
the initial tree

G deleted

A deleted

E deleted

O deleted
B-trees Conclusion

- B-trees are balanced k-ary search trees where $k = 2t$

- The degree of each node is bounded from above and below using the parameter $t$

- All leaves are at the same height

- No rotations are needed: during insertion (or deletion), the balance is maintained by node splitting (or node merging).

- The tree grows (shrinks) in height only by splitting (or merging) the root.