Homework 8 due Tues 11/16

- CLRS 18.1-5 (red-black vs. B-Trees)
- CLRS 18.2-6 (complexity in $t$)
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

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 maximum 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 \( x = \text{root}[T] \).

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

*How do we search for a predecessor?*
B-Tree-Predecessor($T, x, i$)

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

$\text{BDFH} \quad \text{RX}$

$\text{JKL} \quad \text{m} \quad \text{NPQ}$

$\text{BDFH} \quad \text{m} \quad \text{RX}$

$\text{JKL} \quad \text{NPQ}$

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

1: if $n[x] < 2t - 1$ then return “Can’t Split”
2: if $x \neq \text{root}[T]$ and $n[p[x]] = 2t - 1$ then
3: return “Can’t Split”
4: ▷ Create new nodes $y$ and $z$
5: $n[y] \leftarrow t - 1$
6: $n[z] \leftarrow t - 1$
7: for $i \leftarrow 1$ to $t - 1$ do {
8: \hspace{1em} $key_i[y] \leftarrow key_i[x]$
9: \hspace{1em} $key_i[z] \leftarrow key_{i+t}[x]$
10: }
11: for $i \leftarrow 1$ to $t$ do {
12: \hspace{1em} $c_i[y] \leftarrow c_i[x]$
13: \hspace{1em} $c_i[z] \leftarrow c_{i+t}[x]$
14: }
15: ▷ If $x$ is the root then create a new root
16: if $x = \text{root}[T]$ then {
17: \hspace{1em} ▷ Create a new node $v$
18: \hspace{1em} $n[v] \leftarrow 0$
19: \hspace{1em} $c_1[v] \leftarrow x$
20: \hspace{1em} $p[x] \leftarrow v$
21: \hspace{1em} $\text{root}[T] \leftarrow v$
22: }

}
23: \(\triangleright\) 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: \(\triangleright\) 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: \(\quad \text{key}_{i+1}[p[x]] \leftarrow \text{key}_{i}[p[x]]\)
30: \(\quad c_{i+2}[p[x]] \leftarrow c_{i+1}[p[x]]\)
31: \(\}\)
32: \(\triangleright\) Insertion
33: \(\text{key}_j[p[x]] \leftarrow \text{key}_t[x]\)
34: \(c_j[p[x]] \leftarrow y\)
35: \(c_{j+1}[p[x]] \leftarrow z\)
36: \(n[p[x]] \leftarrow n[p[x]] + 1\)
37: \(p[y] \leftarrow p[x]\)
38: \(p[z] \leftarrow p[x]\)
39: \(\triangleright\) 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?
What kind of properties must the input node and the children satisfy for such an operation be possible?

The input node must not be lean, and the two children must be lean.
node x

B D F H m R X

dropped

node y

JKL

node z

NPQ

node x

B D F H R X

node y

JKL m NPQ

node x

B D F H R X

node y

JKL m NPQ
B-Tree-Merge($T, x, i$)

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

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

Before inserting the key we make sure that is room for insertion, that is, not all the nodes in the subtree are full. Since visiting all the nodes in the subtree is very costly, we will make sure only that $y$ 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.
B-Tree-Insert\((T, y, k)\)
1: \(z \leftarrow y\)
2: \(f \leftarrow \text{false}\)
3: \(\text{while } f = \text{false} \text{ do } \{
\)
4: \(\text{if } n[z] = 2t - 1 \text{ then } z \leftarrow \text{B-Tree-Split}(T, z)\)
5: \(j \leftarrow 1\)
6: \(\text{while } \text{key}_j[z] < k \text{ and } j \leq n[z] \text{ do } j \leftarrow j + 1\)
7: \(\text{if } c_j[z] \neq \text{nil} \text{ then } z \leftarrow c_j[z]\)
8: \(\text{else } f \leftarrow \text{true}\)
9: \(\}\)
10: \(\text{for } i \leftarrow n[z] \text{ downto } j \text{ do }\)
11: \(\text{key}_{j+1}[z] \leftarrow \text{key}_j[z]\)
12: \(\text{key}_j \leftarrow k\)
13: \(n[z] \leftarrow n[z] + 1\)
14: \(\text{return } z\)
**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 $k$ in the subtree immediate to the right of $k$. Relocate the predecessor to the position of $k$.

*What should we be careful about?*
We should avoid removing a key from a lean leaf.

To avoid such a case, we can 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.
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.
As we go down the tree, at each level there are three cases:

1. found $k$, this is a leaf

2. found $k$, this is not a leaf

3. did not find $k$

Case 1: $k$ is in a leaf node: remove $k$
Case 2: $k$ is in an internal node:

2a: left child has at least $t$ nodes: take predecessor

2a: right child has at least $t$ nodes: take successor

2c: both children are lean: merge children
Case 3: not found yet: find subtree containing $k$

Case 3a: either sibling has at least $t$ keys: move key

Case 3b: both siblings lean: merge with one sibling
the initial tree

G deleted

A deleted

E deleted

O deleted

GMPX

EMPX

EMPX

MPX

KPX

ACDE JK NO RSTUV YZ

ACD JK NO RSTUV YZ

CD JK NO RSTUV YZ

CDJK NO RSTUV YZ

CDJ MN RSTUV YZ