Homework Solutions – Unit 3: Chapter 18

CMPSC 465

Disclaimer: This is a draft of solutions that has been prepared by the TAs and the instructor makes no guarantees that solutions presented here contain the level of detail that would be expected on an exam. Any errors or explanations you find unclear should be reported to either of the TAs for corrections first.

Exercise 18.1-1

Why don't we allow a minimum degree of $t = 1$?

Solution:

According to the definition, minimum degree t means every node other than the root must have at least $t - 1$ keys, and every internal node other than the root thus has at least t children. So, when $t = 1$, it means every node other than the root must have at least $t - 1 = 0$ key, and every internal node other than the root thus has at least $t = 1$ child.

Thus, we can see that the minimum case doesn't exist, because no node exists with 0 key, and no node exists with only 1 child in a B-tree.

Exercise 18.1-2

For what values of t is the tree of Figure 18.1 a legal B-tree?

Solution:

According to property 5 of B-tree, every node other than the root must have at least $t-1$keys and may contain at most $2t-1$ keys. In Figure 18.1, the number of keys of each node (except the root) is either 2 or 3. So to make it a legal B-tree, we need to guarantee that

$t - 1 \leq 2$ and $2t - 1 \geq 3$,

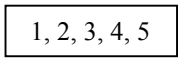
which yields $2 \leq t \leq 3$. So t can be 2 or 3.

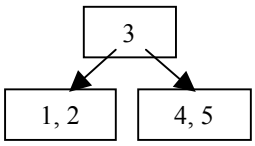
Exercise 18.1-3

Show all legal B-trees of minimum degree 3 that represent {1, 2, 3, 4, 5}.

Solution:

We know that every node except the root must have at least $t - 1 = 2$ keys, and at most $2t - 1 = 5$ keys. Also remember that the leaves stay in the same depth. Thus, there are 2 possible legal B-trees:



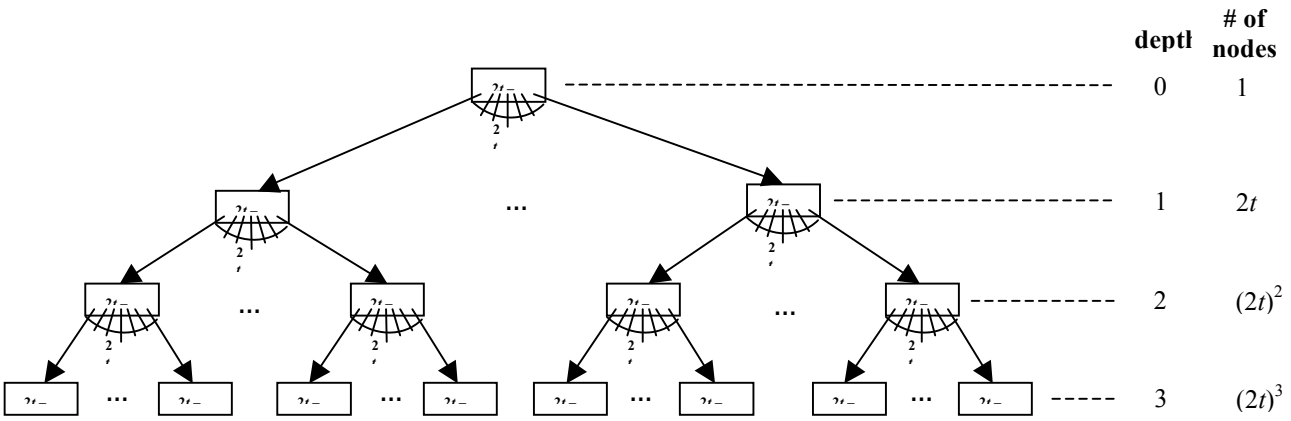


Exercise 18.1-4

As a function of the minimum degree t, what is the maximum number of keys that can be stored in a B-tree of height h?

Solution:

A B-tree with maximum number of keys is shown below:



We can see that each node contains $2t - 1$ keys, and at depth k, the tree at most has $(2t)^k$ nodes. The total nodes is therefore the sum of $(2t)^0$, $(2t)^1$, $(2t)^2$, $(2t)^3$, …, $(2t)^h$. Let MaxKeyNum(t, h) as a function that returns the maximum number of keys in a B-tree of height h and the minimum degree t. We can get that:

$$\begin{aligned}
\text{MaxKeyNum}(t, h) &= (2t-1)[(2t)^0 + (2t)^1 + (2t)^2 + (2t)^3 + \ldots + (2t)^h] && \text{as [keys per node] * [total \# of nodes]} \\
&= (2t-1)\sum_{i=0}^{h}(2t)^i && \text{by using Sigma to sum up total \# of nodes} \\
&= (2t-1)\frac{(2t)^{h+1}-1}{2t-1} && \text{by the summation formula of geometric series} \\
&= (2t)^{h+1}-1
\end{aligned}$$

Exercise 18.1-5

Describe the data structure that would result if each black node in a red-black tree were to absorb its red children, incorporating their children with its own.

Solution:

After absorbing each red node into its black parent, each black node may contain 1, 2 (1 red child), or 3 (2 red children) keys, and all leaves of the resulting tree have the same depth, according to property 5 of red-black tree (For each node, all paths from the node to descendant leaves contain the same number of black nodes). Therefore, a red-black tree will become a B-tree with minimum degree $t = 2$, i.e., a 2-3-4 tree.

Exercise 18.2-3

Explain how to find the minimum key stored in a B-tree and how to find the predecessor of a given key stored in a B-tree.

Solution:

Finding the minimum in a B-tree is quite similar to finding a minimum in a binary search tree. We need to find the left most leaf for the given root, and return the first key.

```
B-TREE-FIND-MIN(x)
//PRE: x is a node on the B-tree T. The top level call is B-TREE-FIND-MIN(T.root).
//POST: FCTVAL is the minimum key stored in the subtree rooted at x.
{
    if x == NIL                          //T is empty
    {
        return NIL
    }
    else if x.leaf                       //x is leaf
    {
        return x.key_1                   //return the minimum key of x
    }
    else
    {
        DISK-READ(x.c_1)
        return B-TREE-FIND-MIN(x.c_1)
    }
}
```

Finding the predecessor of a given key $x.key_i$ is according to the following rules:

- If x is not a leaf, return the maximum key in the i-th child of x, which is also the maximum key of the subtree rooted at $x.c_i$
- If x is a leaf and $i > 1$, return the $(i–1)$st key of x, i.e., $x.key_{i–1}$
- Otherwise, look for the last node y (from the bottom up) and $j > 0$, such that $x.key_i$ is the leftmost key in $y.c_j$; if j = 1, return NIL since $x.key_i$ is the minimum key in the tree; otherwise we return $y.key_{j–1}$.

```
B-TREE-FIND-PREDECESSOR(x, i)
//PRE: x is a node on the B-tree T. i is the index of the key.
//POST: FCTVAL is the predecessor of x.key_i.
{
    if ! x.leaf
    {
        DISK-READ(x.c_i)
        return B-TREE-FIND-MAX(x.c_i)
    }
    else if i > 1                        //x is a leaf and i > 1
        return x.key_{i–1}
    else                                 //x is a leaf and i = 1
    {
        z = x

        while (1)
        {
            if z.p == NIL                //z is root
                return NIL               // z.key_i is the minimum key in T; no predecessor

            y = z.p
            j = 1
            DISK-READ(y.c_1)
```

```
            while (y.c_j != x)
            {
                j = j + 1
                DISK-READ(y.c_j)
            }

            if j == 1
                z = y
            else
                return y.key_{j-1}
        }
    }
}
```

```
B-TREE-FIND-MAX(x)
//PRE: x is a node on the B-tree T. The top level call is B-TREE-FIND-MAX(T.root).
//POST: FCTVAL is the maximum key stored in the subtree rooted at x.
{
    if x == NIL                                  //T is empty
    {
        return NIL
    }
    else if x.leaf                               //x is leaf
    {
        return (x, x.n)                          //return the maximum key of x
    }
    else
    {
        DISK-READ(x.c_{x.n+1})
        return B-TREE-FIND-MAX(x.c_{x.n+1})
    }
}
```

Exercise 18.2-6

Suppose that we were to implement B-TREE-SEARCH to use binary search rather than linear search within each node. Show that this change makes the CPU time required $O(\lg n)$, independently of how t might be chosen as a function of n.

Solution:

As in the TREE-SEARCH procedure for binary search trees, the nodes encountered during the recursion form a simple path downward from the root of the tree. Thus, the B-TREE-SEARCH procedure needs $O(h) = O(\log_t n)$ CPU time to search along the path, where h is the height of the B-tree and n is the number of keys in the B-tree, and we know that $h \leq \log_t \frac{n+1}{2}$. Since the number of keys in each node is less than $2t - 1$, a binary search within each node is $O(\lg t)$. So the total time is:

$$O(\lg t * \log_t n) = O(\lg t * \frac{\lg n}{\lg t}) \quad \text{by changing the base of the logarithm}$$
$$= O(\lg n)$$

Thus, the CPU time required is $O(\lg n)$.