18 B-Trees

18.1 Definition of B-trees

18.1-1

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

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.

18.1-2

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

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 $t - 1 \le 2 \text{ and } 2t - 1 \ge 3$, which yields $2 \le t \le 3$. So t can be 2 or 3.

18.1-3

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

We know that every node except the root must have at least $t - 1 = 1$ keys, and at most $2t - 1 = 3$ keys. Also remember that the leaves stay in the same depth. Thus, there are 2 possible legal B-trees:

- $$|1, 2, 3, 4, 5|$$
- $$|3|$$
  $$\swarrow \quad \searrow$$
  $$|1, 2| \qquad\qquad |4, 5|$$

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?

$$\begin{aligned} n &= (1 + 2t + (2t)^2 + \cdots + (2t)^h) \cdot (2t - 1) \\ &= (2t)^{h+1} - 1. \end{aligned}$$

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.

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 Btree with minimum degree $t = 2$, i.e., a 2-3-4 tree.

18.2 Basic operations on B-trees

18.2-1

> Show the results of inserting the keys
>
> $$F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$$
>
> in order into an empty B-tree with minimum degree 2. Draw only the configurations of the tree just before some node must split, and also draw the final configuration.

(Omit!)

18.2-2

> Explain under what circumstances, if any, redundant DISK-READ or DISK-WRITE operations occur during the course of executing a call to B-TREE-INSERT. (A redundant DISK-READ is a DISK-READ for a page that is already in memory. A redundant DISK-WRITE writes to disk a page of information that is identical to what is already stored there.)

In order to insert the key into a full child node but without its parent being full, we need the following operations:

- DISK-READ: Key placement
- DISK-WRITE: Split nodes
- DISK-READ: Get to the parent
- DISK-WRITE: Fill parent

If both were full, we'd have to do the same, but instead of the final step, repeat the above to split the parent node and write into the child nodes. With both considerations in mind, there should never be a redundant DISK-READ or DISK-WRITE on a B-TREE-INSERT.

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.

- 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.
    - **PRE:** x is a node on the B-tree T. The top level call is $\text{B-TREE-FIND-MIN}(T.root)$.
    - **POST:** FCTVAL is the minimum key stored in the subtree rooted at x.

    ```
    B-TREE-FIND-MIN(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:

    1. 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 $x.c_i$.

    2. If x is a leaf and $i > 1$, return the $(i-1)$st key of x, i.e., $x.key_{i-1}$.

    3. 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 $y.key_{j-1}$.

        - **PRE:** x is a node on the B-tree T. i is the index of the key.
        - **POST:** FCTVAL is the predecessor of $x.key_i$.

    ```
    B-TREE-FIND-PREDECESSOR(x, 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
            z = x
            while true
                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]
```

- **PRE:** x is a node on the B-tree T. The top level call is $\text{B-TREE-FIND-MAX}(T.\text{root})$.
- **POST:** FCTVAL is the maximum key stored in the subtree rooted at x.

```
B-TREE-FIND-MAX(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-MIN(x.c[x.n + 1])
```

18.2-4 ⋆

> Suppose that we insert the keys $\{1, 2, \ldots, n\}$ into an empty B-tree with minimum degree 2. How many nodes does the final B-tree have?

The final tree can have as many as $n - 1$ nodes. Unless $n = 1$ there cannot ever be n nodes since we only ever insert a key into a non-empty node, so there will always be at least one node with 2 keys.

Next observe that we will never have more than one key in a node which is not a right spine of our B-tree. This is because every key we insert is larger than all keys stored in the tree, so it will be inserted into the right spine of the tree. Nodes not in the right spine are a result of splits, and since $t = 2$, every split results in child nodes with one key each. The fewest possible number of nodes occurs when every node in the right spine has 3 keys. In this case, $n = 2h + 2^{h+1} - 1$ where h is the height of the B-tree, and the number of nodes is $2^{h+1} - 1$. Asymptotically these are the same, so the number of nodes is $\Theta(n)$.

18.2-5

> Since leaf nodes require no pointers to children, they could conceivably use a different (larger) t value than internal nodes for the same disk page size. Show how to modify the procedures for creating and inserting into a B-tree to handle this variation.

You would modify the insertion procedure by, in B-TREE-INSERT, check if the node is a leaf, and if it is, only split it if there twice as many keys stored as expected. Also, if an element needs to be inserted into a full leaf, we would split the leaf into two separate leaves, each of which doesn't have too many keys stored in it.

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.

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 $h \le \log_t \frac{n+1}{2}$. Since the number of keys in each nodeis less than $2t - 1$, a binary search within each node is $O(\lg t)$. So the total time is:

$$\begin{aligned} O(\lg t \cdot \log_t n) &= O(\lg t \cdot \frac{\lg n}{\lg t}) && \text{by changing the base of the logarithm.} \\ &= O(\lg n). \end{aligned}$$

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

18.2-7

> Suppose that disk hardware allows us to choose the size of a disk page arbitrarily, but that the time it takes to read the disk page is $a + bt$, where a and b are specified constants and t is the minimum degree for a B-tree using pages of the selected size. Describe how to choose t so as to minimize (approximately) the B-tree search time. Suggest an optimal value of t for the case in which $a = 5$ milliseconds and $b = 10$ microseconds.

$$\min \log_t n \cdot (a + bt) = \min \frac{a + bt}{\ln t}$$

$$\frac{\partial}{\partial t}(\frac{a + bt}{\ln t}) = -\frac{a + bt - bt \ln t}{t \ln^2 t}$$

$$\begin{aligned} a + bt &= bt \ln t \\ 5 + 10t &= 10t \ln t \\ t &= e^{W\left(\frac{1}{2e}\right) + 1}, \end{aligned}$$

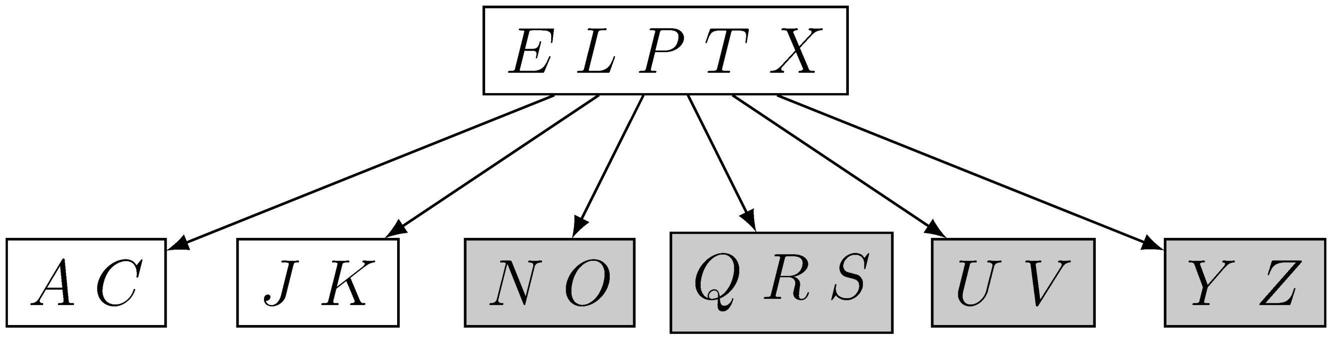
where W is the LambertW function, and we should choose $t = 3$.

18.3 Deleting a key from a B-tree

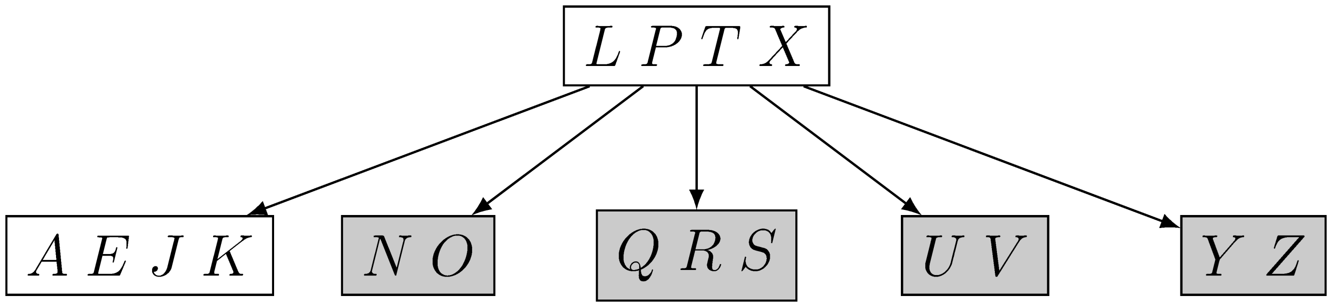
18.3-1

> Show the results of deleting C, P, and V, in order, from the tree of Figure 18.8(f).

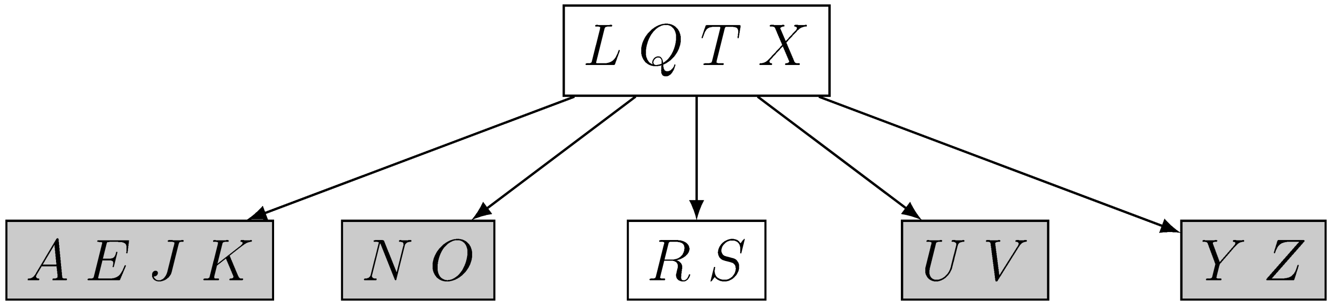
- Figure 18.8(f)



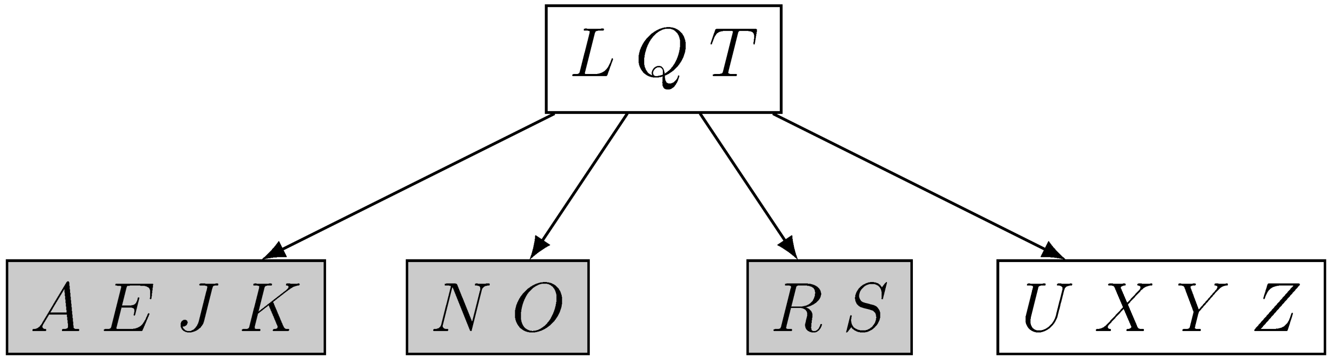
- delete C



- delete P



- delete V



18.3-2

> Write pseudocode for B-TREE-DELETE.

The algorithm B-TREE-DELETE(x, k) is a recursive procedure which deletes key k from the B-tree rooted at node x.

The functions $\text{PREDECESSOR}(k, x)$ and $\text{SUCCESSOR}(k, x)$ return the predecessor and successor of k in the B-tree rooted at x respectively.

The cases where k is the last key in a node have been omitted because the pseudocode is already unwieldy. For these, we simply use the left sibling as opposed to the right sibling, making the appropriate modifications to the indexing in the **for** loops.

Problem 18-1 Stacks on secondary storage

> Consider implementing a stack in a computer that has a relatively small amount of fast primary memory and a relatively large amount of slower disk storage. The operations PUSH and POP work on single-word values. The stack we wish to support can grow to be much larger than can fit in memory, and thus most of it must be stored on disk.
>
> A simple, but inefficient, stack implementation keeps the entire stack on disk. We maintain in - memory a stack pointer, which is the disk address of the top element on the stack. If the pointer has value p, the top element is the $(p \mod m)$th word on page $\lfloor p/m \rfloor$ of the disk, where m is the number of words per page.
>
> To implement the PUSH operation, we increment the stack pointer, read the appropriate page into memory from disk, copy the element to be pushed to the appropriate word on the page, and write the page back to disk. A POP operation is similar. We decrement the stack pointer, read in the appropriate page from disk, and return the top of the stack. We need not write back the page, since it was not modified.
>
> Because disk operations are relatively expensive, we count two costs for any implementation: the total number of disk accesses and the total CPU time. Any disk access to a page of m words incurs charges of one disk access and $\Theta(m)$ CPU time.
>
> **a.** Asymptotically, what is the worst-case number of disk accesses for n stack operations using this simple implementation? What is the CPU time for n stack operations? (Express your answer in terms of m and n for this and subsequent parts.)
>
> Now consider a stack implementation in which we keep one page of the stack in memory. (We also maintain a small amount of memory to keep track of which page is currently in memory.) We can perform a stack operation only if the relevant disk page resides in memory. If necessary, we can write the page currently in memory to the disk and read in the new page from the disk to memory. If the relevant disk page is already in memory, then no disk accesses are required.
>
> **b.** What is the worst-case number of disk accesses required for n PUSH operations? What is the CPU time?
>
> **c.** What is the worst-case number of disk accesses required for n stack operations? What is the CPU time?

> Suppose that we now implement the stack by keeping two pages in memory (in addition to a small number of words for bookkeeping).
>
> **d.** Describe how to manage the stack pages so that the amortized number of disk accesses for any stack operation is $O(1/m)$ and the amortized CPU time for any stack operation is $O(1)$.

a. We will have to make a disk access for each stack operation. Since each of these disk operations takes time $\Theta(m)$, the CPU time is $\Theta(mn)$.

b. Since only every mth push starts a new page, the number of disk operations is approximately n/m, and the CPU runtime is $\Theta(n)$, since both the contribution from the cost of the disk access and the actual running of the push operations are both $\Theta(n)$.

c. If we make a sequence of pushes until it just spills over onto the second page, then alternate popping and pulling many times, the asymptotic number of disk accesses and CPU time is of the same order as in part a. This is because when we are doing that alternating of pops and pushes, each one triggers a disk access.

d. We define the potential of the stack to be the absolute value of the difference between the current size of the stack and the most recently passed multiple of m. This potential function means that the initial stack which has size 0, is also a multiple of m, so the potential is zero. Also, as we do a stack operation we either increase or decrease the potential by one. For us to have to load a new page from disk and write an old one to disk, we would need to be at least m positions away from the most recently visited multiple of m, because we would have had to just cross a page boundary. This cost of loading and storing a page takes (real) cpu time of $\Theta(m)$. However, we just had a drop in the potential function of order $\Theta(m)$. So, the amortized cost of this operation is $O(1)$.

Problem 18-2 Joining and splitting 2-3-4 trees

> The ***join*** operation takes two dynamic sets S' and S'' and an element x such that for any $x' \in S'$ and $x'' \in S''$, we have $x'.key < x.key < x''.key$. It returns a set $S = S' \cup \{x\} \cup S''$. The ***split*** operation is like an "inverse" join: given a dynamic set S and an element $x \in S$, it creates a set S' that consists of all elements in set S and an element $x \in S$, it creates a set S' that consists of all elements in $S - \{x\}$ whose keys are less than $x.key$ and a set S'' that consists of all elements in $S - \{x\}$ whose keys are greater than $x.key$. In this problem, we investigate how to implement these operations on 2-3-4 trees. We assume for convenience that elements consist only of keys and that all key values are distinct.
>
> **a.** Show how to maintain, for every node x of a 2-3-4 tree, the height of the subtree rooted at x as an attribute $x.height$. Make sure that your implementation does not affect the asymptotic running times of searching, insertion, and deletion.
>
> **b.** Show how to implement the join operation. Given two 2-3-4 trees T' and T'' and a key k, the join operation should run in $O(1 + |h' - h''|)$ time, where h' and h'' are the heights of T' and T'', respectively.

> **c.** Consider the simple path p from the root of a 2-3-4 tree T to a given key k, the set S' of keys in T that are less than k, and the set S'' of keys in T that are greater than k. Show that p breaks S' into a set of trees $\{T'_0, T'_1, \ldots, T'_m\}$ and a set of keys $\{k'_1, k'_2, \ldots, k'_m\}$, where, for $i = 1, 2, \ldots, m$, we have $y < k'_i < z$ for any keys $y \in T'_{i-1}$ and $z \in T'_i$. What is the relationship between the heights of T'_{i-1} and T'_i? Describe how p breaks S'' into sets of trees and keys.
>
> **d.** Show how to implement the split operation on T. Use the join operation to assemble the keys in S' into a single 2-3-4 tree T' and the keys in S'' into a single 2-3-4 tree T''. The running time of the split operation should be $O(\lg n)$, where n is the number of keys in T. $($Hint: The costs for joining should telescope.)

a. For insertion it will suffice to explain how to update height when we split a node. Suppose node x is split into nodes y and z, and the median of x is merged into node w. The height of w remains unchanged unless x was the root (in which case $w.\text{height} = x.\text{height} + 1$).

The height of y or z will often change. We set

$$y.\text{height} = \max_i y.c_i.\text{height} + 1$$

and

$$z.\text{height} = \max_i z.c_i.\text{height} + 1.$$

Each update takes $O(t)$. Since a call to B-TREE-INSERT makes at most h splits where h is the height of the tree, the total time it takes to update heights is $O(th)$, preserving the asymptotic running time of insert. For deletion the situation is even simple. The only time the height changes is when the root has a single node and it is merged with its subtree nodes, leaving an empty root node to be deleted. In this case, we update the height of the new node to be the (old) height of the root minus 1.

b. Without loss of generality, assume $h' \ge h''$. We essentially wish to merge T'' into T' at a node of height h'' using node x. To do this, find the node at depth $h' - h''$ on the right spine of T'. Add x as a key to this node, and T'' as the additional child. If it should happen that the node was already full, perform a split operation.

c. Let x_i be the node encountered after i steps on path p. Let l_i be the index of the largest key stored in x_i which is less than or equal to k. We take $k'_i = x_i.\text{key}_{l_i}$ and T'_{i-1} to be the tree whose root node consists of the keys in x_i which are less than $x_i.\text{key}_{l_i}$, and all of their children. In general, $T'_{i-1}.\text{height} \ge T'_i.\text{height}$.

For S'', we take a similar approach. They keys will be those in nodes passed on p which are immediately greater than k, and the trees will be rooted at a node consisting of the larger keys, with the associated subtrees. When we reach the node which contains k, we don't assign a key, but we do assign a tree.

d. Let T_1 and T_2 be empty trees. Consider the path p from the root of T to k. Suppose we have reached node x_i. We join tree T'_{i-1} to T_1, then insert $k'i$ into T_1. We join T''_{i-1} to T_2 and insert k''_i into T_2. Once we have encountered the node which contains k at $x_m.\text{key}_k$, join $x_m.c_k$ with T_1 and $x_m.c_{k+1}$ with T_2.

We will perform at most 2 join operations and 1 insert operation for each level of the tree. Using the runtime determined in part (b), and the fact that when we join a tree T' to T_1 (or T'' to T_2 respectively) the height difference is

$$T'.\text{height} - T_1.\text{height}.$$

Since the heights are nondecreasing of successive tree that are joined, we get a telescoping sum of heights. The first tree has height h, where h is the height of T, and the last tree has height 0. Thus, the runtime is

$$O(2(h + h)) = O(\lg n).$$