Lecture Notes for CSCI 311: Algorithms

Set 14-B Trees

Professor Talmage

October 16, 2026

---

1 Balanced Search Trees

Exercise: What are the runtimes of BST operations (particularly insert, delete, search)?

These operations all run in the height of tree, worst case, which can be $O(n)$ in a BST with n nodes. We want to *balance* the tree, meaning that about half of the nodes in each subtree will be on each side of that subtree's root. This means that the tree's height is $O(\log n)$, so all operations run in $O(\log n)$ time. There are many ways to balance a BST. You may have seen AVL trees in CSCI 204, which are a simple way to do this.

One more advanced way to build a BST is a *Red-Black Tree*. These store all data in internal nodes, using leaves as dummy nodes to help track balance. Red-Black Trees maintain the invariant that for every leaf x in tree T, $x.depth \geq T.height/2$, meaning the tree is fairly densely packed. This leads to a proof that $T.height \leq 2\log(n+1) = O(\log n)$. To maintain this height, the tree maintains a coloring scheme where each node is either red or black. These colors tell us when we need to rebalance, but insertion and deletion wind up being fairly complex, as we have to rotate entire subtrees to put more nodes on the shorter side, while also maintaining the coloring properties. We will talk about these trees more in recitation, but the important takeaway is that it is possible to maintain a balanced BST in $O(\log n)$ time per operation.

2 B-Trees

We will spend more time on *B-Trees*, which are a different type of balanced search tree, though they are not binary, meaning a node can have more than two children. These are not somehow "better" than Red-Black trees, but we do not have time to cover both in detail. In both cases, your takeaway should be that we can efficiently maintain search trees of small height.

Motivation: As mentioned, we want a tree with height $O(\log n)$, but what if accessing each node is **very** expensive? B-Trees were originally developed for use in file systems, which have to deal with much slower storage disks than the RAM programs typically use. On the other hand, storage devices typically return an entire page (something like 30KB) for the same time cost as reading a single bit.

Aside: Hard Disk Drives are typically about 5 orders of magnitude slower than RAM. Solid-state drives have improved storage speed greatly, but are still slower than RAM. Some comparisons:

- My computer's RAM runs at 3200MT/s (million transfers per second). Some web searching[1]suggests I could theoretically get up to about 51 GB/s. The fastest RAM I see for sale now is 9000 MT/s, which would more than triple the bandwidth, to 144 GB/s. (Of course, I cannot simply upgrade to that without upgrading my motherboard and processor...)
- I have an NVMe 3 SSD, which is advertised at 3000 MB/s write, 3400 MB/s read. Several companies are currently advertising high-performance SSDs around 14,000 MB/s write, 12,000 MB/s read (though random access is much slower).
- This is roughly a 15x advantage of RAM over storage in my machine or more than a 10x advantage using top-end components.
- Note that these are theoretical capabilities. Under typical workloads, storage is **much** slower than advertised speeds, while RAM is fairly close. Also, unless you are using a lot of memory, your program will primarily be running in CPU cache, which is far faster than RAM, increasing the disparity.

The idea for an efficient tree given these constraints is to have many children/node to decrease tree height. Then, since each read gives a large block of memory, put many keys in each node to use that space. The goal is for a single node in our tree to fill an entire page on disk. Of course, with many keys per node, the BST Property no longer makes sense, so we need to define a new search tree property.

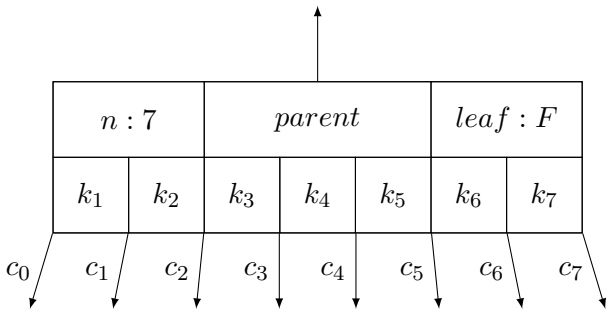
Definition 1. A B-Tree is a rooted tree with five properties:

1. Every node x contains
   (a) $x.n$: number of keys in x
   (b) $x.n$ keys in non-decreasing order
   (c) $x.leaf$: Boolean indicating whether this node is a leaf
   (d) $x.parent$ (optional)
2. Every internal node x has $x.n + 1$ child pointers.
3. The keys in a node divide the ranges of keys in the node's subtrees.
   - Search Tree Property: All keys in the subtree at child 0 are less than or equal to key 0, which is less than or equal to all keys in the subtree rooted at child 1, which are less than or equal to key 1, ..., key n is less than or equal to all keys in the subtree rooted at child n.
4. All leaves are at the same depth.
5. A B-Tree has a parameter t, called its *minimum degree*, s.t.
   (a) Every non-root node has at least $t - 1$ keys (root has at least 1)
   (b) Every node has at most $2t - 1$ keys

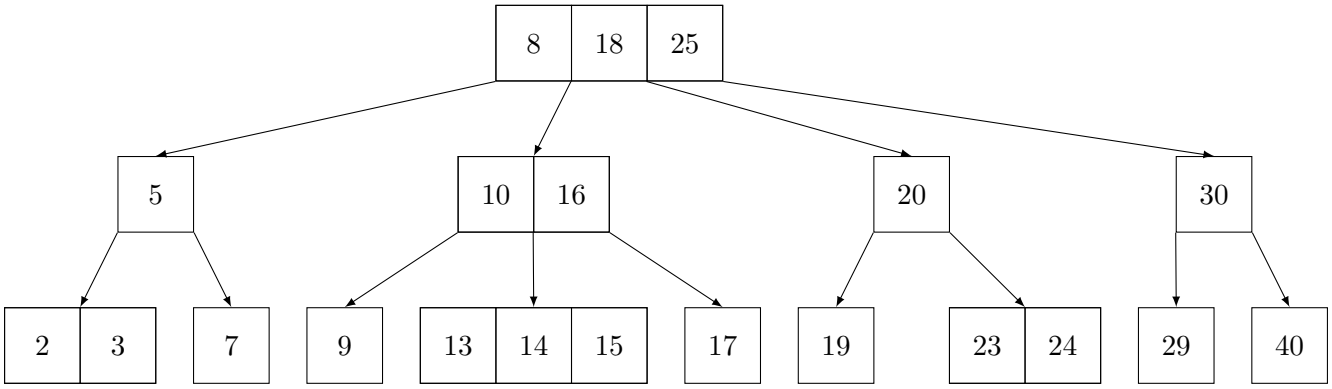
   - Keys/node is in $[t - 1, 2t - 1]$, so children/node is in $[t, 2t]$.
   - If $t = 2$, every non-root node has 2, 3, or 4 children, and this is sometimes called a 2-3-4 tree.

[1] `https://edu.finlaydag33k.nl/calculatingrambandwidth/`

A single node looks something like this:



When drawing an entire B-Tree, it is often easier to simplify notation somewhat:



Exercise: Insert the following values to this B-Tree with minimum degree $t = 2$: $1, 27, 6, 26, 28, 12, 11, 10.5$. Speculate how we will maintain B-Tree properties when inserting into a full node.

We can now prove that B-Trees have low height, which will ensure that the runtimes of functions on them is also low.

Theorem 1. *If $n \geq 0$, then for any n-key B-Tree of height h and min-degree $t \geq 2$, $h \leq \log_t\left(\frac{n+1}{2}\right)$.*

Proof. If $n = 0$, then *height* $= 0$ and we are done. Otherwise, the root has at least one value. If $h > 1$, this means that there are at least 2 nodes at depth 1. Recall that every non-root node has at least $t - 1$ values, implying it has at least t children. Thus, there are at least 2 nodes at depth 1, $2t$ at depth 2, and in general $2t^{d-1}$ at depth d, if the tree height is at least d.

We then have that the total number of keys, n, must satisfy

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

If we solve this for h, we find that $h \leq \log_t\left(\frac{n+1}{2}\right)$. □

Aside: To expand the sum in the above proof, note that the $t-1$ factor creates two shifted copies of the sum of powers of t, which mostly cancel out.

$$\sum_{i=1}^{h} t^{i-1} = 1 + t + t^2 + \cdots + t^{h-1}$$

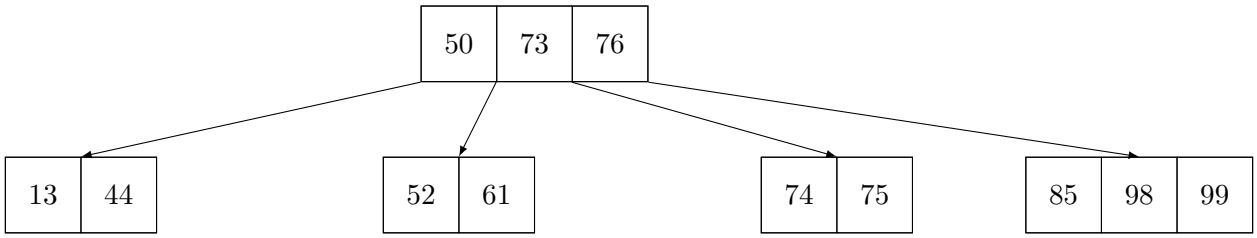
$$(t-1)\sum_{i=1}^{h} t^{i-1} = (t + t^2 + t^3 + \cdots + t^h) - (1 + t + t^2 + \cdots + t^{h-1})$$

$$= t^h - 1$$

In general, $a^n - b^n = (a-b)(a^{n-1} + a^{n-2}b + \cdots + ab^{n-2} + ab^{n-1})$.

Exercise: Draw a B-Tree with minimum degree $t = 3$ containing the values $[98, 74, 50, 13, 44, 75, 76, 73, 52, 85, 61, 99]$. Compare with a neighbor and discuss any differences in your trees.

One possible tree (there are several variations on this):



3 Operations

We want to implement the following operations:

- Accessors:
  - `BT-Search`$(root, key)$
  - `BT-Min`$(root)$
  - `BT-Max`$(root)$
- Mutators:
  - `BT-Insert`(key)
  - `BT-Delete`$(root, key)$
  - `BT-Split`$(parent, i)$
  - `BT-InsertNonfull`$(root, key)$

3.1 Searching

Exercise: Generalize `BST-Search` to `BT-Search`.

```
1: function BT-Search(root, key)
2:     i = 1
3:     while i ≤ root.n and key > root.key_i do
4:         i = i + 1
```

```
    end while
    if i ≤ root.n and key == root.key_i then
        return (root, i)
    else if root is a leaf then
        return ⊥
    else
        Disk-Read(root.child_{i−1})
        return BT-Search(root.child_{i−1}, key)
    end if
end function
```

Complexity Linear search in a node is $O(t)$, since there are at most $2t-1$ keys per node. We prove that the the tree's height is $O(\log_t n)$, so we have only that many nodes to search, since we consider a single node at each level. This gives a total time complexity of $O(t \log n)$.

Exercise: What if we use binary search among the keys within each node? Does this improve our runtime?

- Since t is constant in the tree, we end up with the same asymptotic runtime as linear search.
- If t is very large, there may be some practical improvement.

For space complexity, each call must read one node from disk, so we get a total of $h + 1 = O(\log_t n)$ disk accesses.

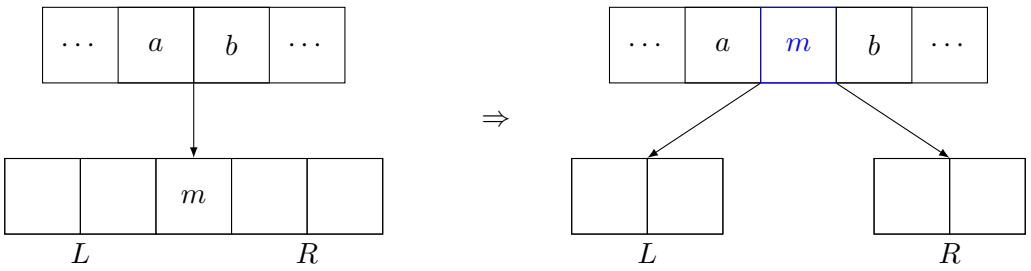
3.2 Insertion

Similarly to in BSTs, we will search to find the correct place to insert, then add the new key. This means that we always add new keys to leaves. The trick, as we saw earlier, is to not break the B-Tree properties. Specifically,

- We cannot create a new node with only one element (unless $t = 2$, but must still respect key-child counts).
- We must keep all leaves at the same depth, so cannot add the key below an existing leaf.
- We do not want to overfill a node, so cannot insert into a full node.

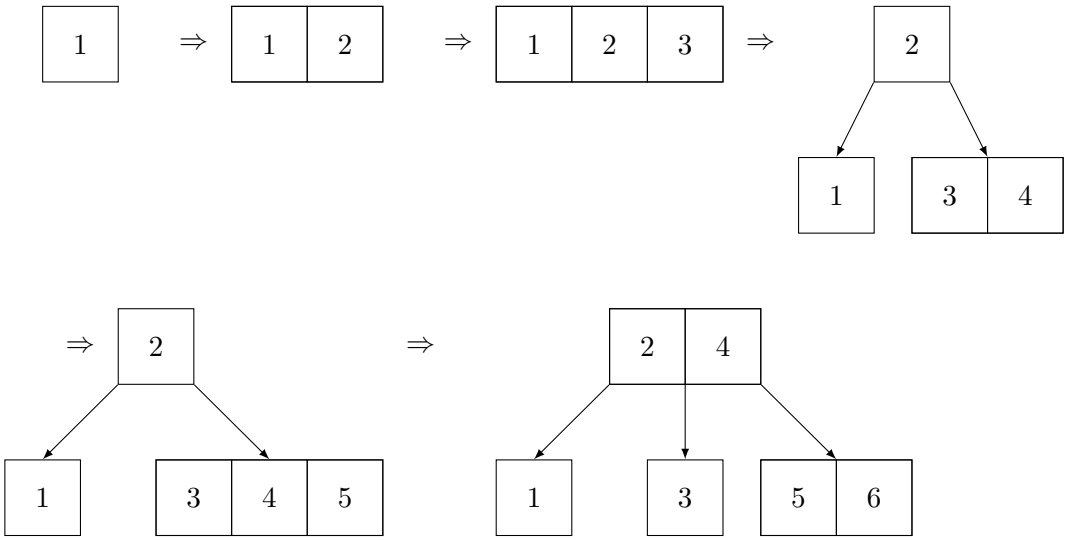
To handle the first two constraints, we always insert into an existing leaf, in proper key order. For the third constraint, we need to ensure that we never enter a full leaf with intent to insert a value into it. To do this, we must split full leaves before entering them. A node is full when it contains $2t-1$ keys. To split such a node:

1. Identify the leaf into which we will insert our new key.
2. Promote full leaf's median key into parent node.
3. Connect remaining $2(t-1)$ keys as two new leaves, on each side of promoted key.

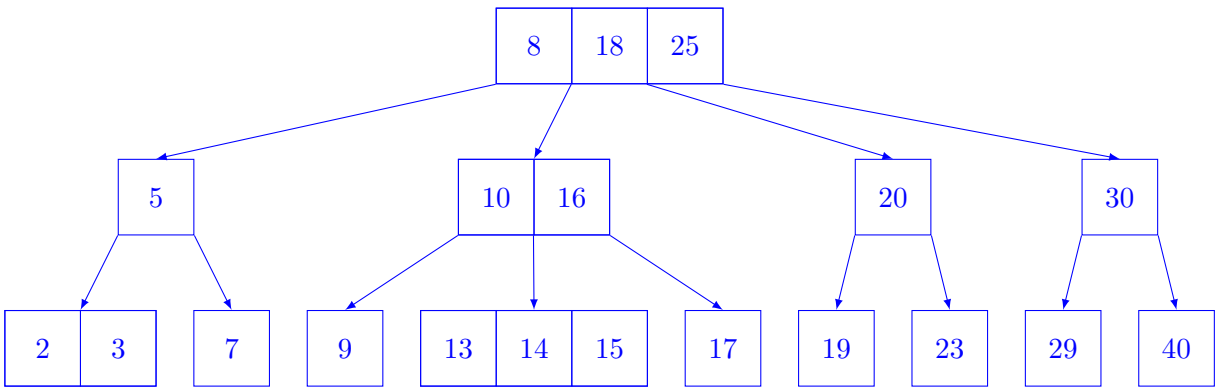


The problem with this approach is that if the leaf's parent is also full, then we cannot promote the median key without splitting the parent. But if that node's parent is full, we have the same problem, and so on up the tree. This could lead to a full two traversals of the height of the tree, loading each node on that path into memory twice. To prevent this, we will proactively refuse to enter any full node, not just the leaf where we might insert the node. By always having extra space, if the next node we need to visit is full, we can split it, promoting the median into our current (non-full) node. Then we can safely enter the next node. This aggressive splitting guarantees that we will have space in the leaf we finally reach, and prevents us having to traverse back up the tree (extra disk accesses!) to make space.

Example: Insert $1, 2, 3, 4, 5, 6$ into an initially empty B-Tree with $t = 2$:



Exercise: Insert three new values into the below B-Tree with minimum degree $t = 2$. Try to choose values that will end up in different parts of the tree.



Pseudocode

```
1: function BT-Insert(x)                          ▷ Wrapper to ensure root not full
2:     r = root
3:     if r is full then
4:         Allocate new, empty node s
5:         s.c_1 = r
6:         BT-Split(s, 1)                          ▷ Helper to split child 1 of non-full node s
7:         root = s
```

```
    end if
    BT-InsertNonfull(root, x)
end function

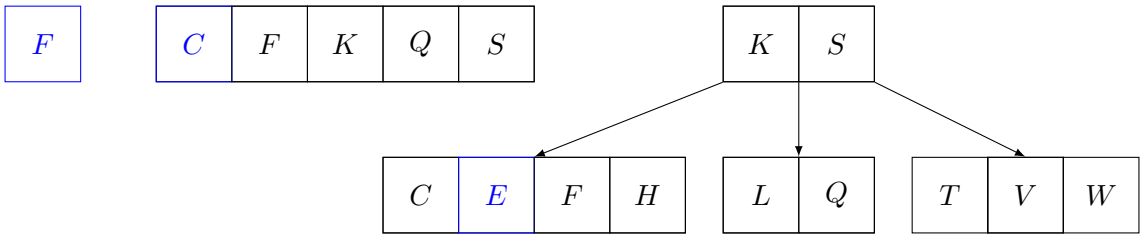
function BT-InsertNonfull(root, x)                    ▷ Precondition: root is not full
    if root.leaf then
        insert x as a key in root in sorted order
        write root to disk
    end if
    i = root.n
    while i ≥ 1 and x < root.key_i do
        i = i − 1
    end while
    i = i + 1
    read node root.child_i from disk
    if root.child_i is full then
        BT-Split(root, i)
        if x > root.key_i then i = i + 1
        end if
    end if
    BT-InsertNonfull(root.child_i, x)
end function

function BT-Split(parent, i)
    move median of parent.child_i to be parent.key_i, adjusting indexing of other keys and child pointers in parent
    split parent.child_i (excluding median) into two nodes, n_1 and n_2, each with t − 1 keys (allocate one new node)
    parent.child_i = n_1
    parent.child_{i+1} = n_2
end function
```

Complexity:

- `BT-Split`: $O(t)$ time, $O(1)$ disk accesses.
- `BT-InsertNonfull`:
  - Searching in a node: $O(t)$ time per call
  - Split: $O(t)$ time, $O(1)$ disk per call
  - $O(h) = O(\log_t n)$ recursive calls
  - Total $O(t)O(h) = O(t \log_t n)$ time, $h + 1 = O(\log_t n)$ disk
- `BT-Insert`: Cost of `BT-InsertNonfull` plus at most one split, $O(t \log_t n)$ time, $h + 1 = O(\log_t n)$ disk

Exercise: Insert the following keys, in order, into an initially empty B-Tree with $t = 3$. Keys: $F, S, Q, K, C, L, H, T, V, W, E$. Use alphabetical order.

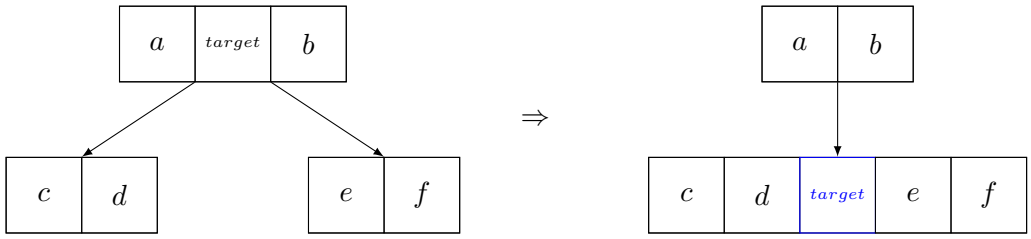


3.3 Deletion

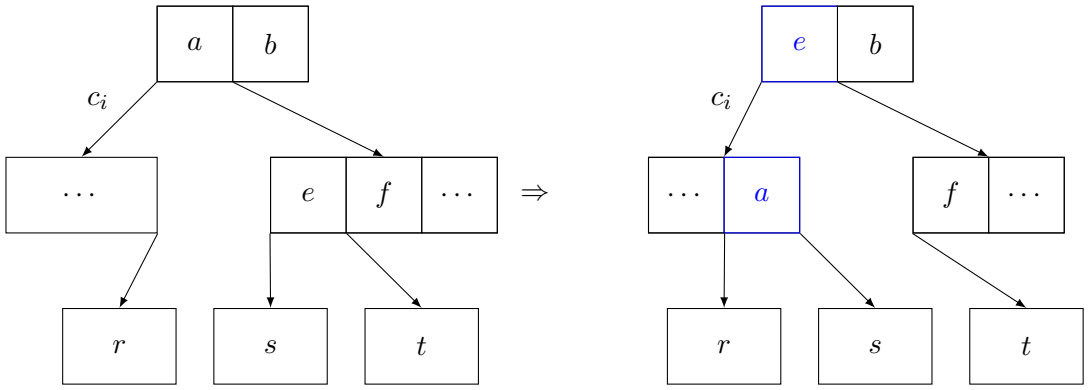
Similarly, but symmetrically, to our concerns with insertion, when deleting we need to be careful about shrinking nodes below minimum size, and we do not want to strand children, since removing a key decreases the number of children a node can have. Our solution is also similar. We will proactively merge minimum-size nodes on our way down the tree to ensure we are never in a node which cannot shrink. This lets us complete our delete in a single downward pass through the tree.

As we recursively travel down the tree, we have three cases for the current node: (1) we are in a leaf, (2) we are in an internal node containing the key to delete, (3) we are in an internal node not containing the target key. We handle each of these cases separately:

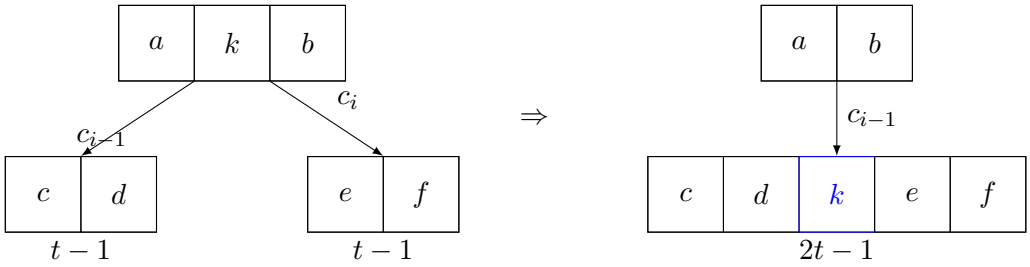
1. If in leaf, delete target directly. If it is not here, it is not in the tree, so do nothing.
2. If in internal node *root* containing target key:
   (a) If $root.child_i$ (child before target) has at least t keys, find $p = predecessor(target)$, recursively delete p, replace target with p.
   (b) Symmetrically, if $root.child_{i+1}$ has at least t keys, replace target with $successor(target)$.
   (c) If both of the target key's adjacent children are minimum size, merge the target down with both of them to create a maximum size node, then recursively descend into that node.



3. If in internal node *root* not containing target key:
   (a) Determine which child $root.child_i$ must contain target
   (b) Ensure that $root.child_i$ is not minimum size
      i. If $root.child_i$ has only $t-1$ keys, and has an immediate sibling ($root.child_{i\pm1}$) with at least t keys, move one of those extra keys up to *root*, moving the intermediate key down into $root.child_i$, adjusting child pointers. This is a form of rotation, though perhaps simpler than that in AVL or Red-Black Trees.



ii. If $root.child_i$ and both of its immediate siblings have only $t-1$ keys, merge $child_i$ with one of its neighbors, bringing the intermediate key down from *root* to be the new median.

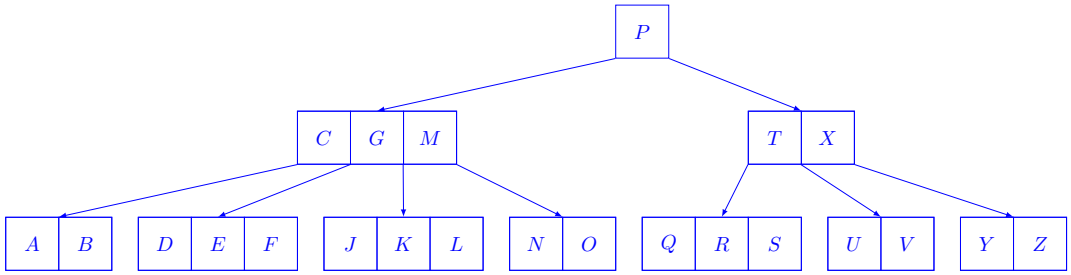


(c) Recursively call `BT-Delete` on $root.child_i$.

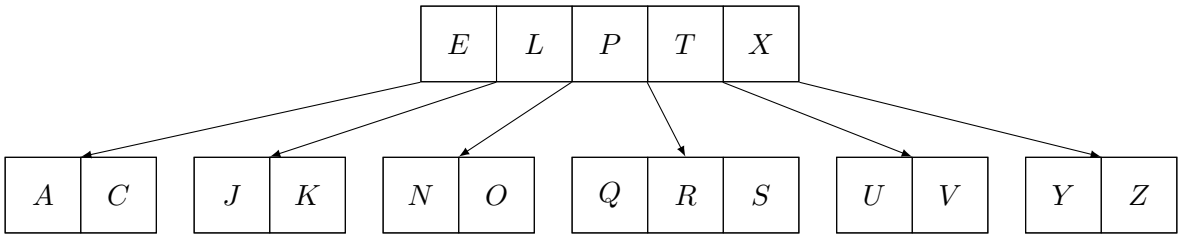
Complexity: We need only a single downward pass to find the target key. We may need to continue downward from there to find the predecessor/successor, but overall a single downward pass. This gives a total time of $O(t \log_t n)$, total number of disk accesses $O(\log_t n)$.

Exercise: Execute the following sequence of calls on the given B-Tree with $t = 3$:

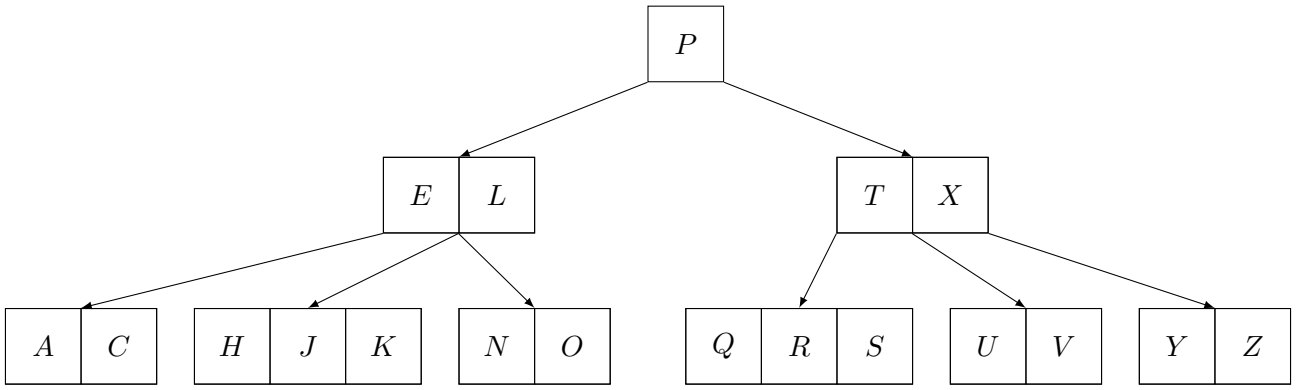
$$Delete(F) \cdot Delete(M) \cdot Delete(G) \cdot Delete(D) \cdot Delete(B) \cdot Insert(H)$$



After deletions, the tree looks like this:



And finally, after all operation instances:


P
E
L
T
X
A
C
H
J
K
N
O
Q
R
S
U
V
Y
Z