Chapter 18: B-Trees

Data Structures on Secondary Storage

Databases are stored on disk (as multiple files) so that we can persist our data beyond the running of the programs. The database software stores the data structures in the files on disk.

Sometimes our data structures are too big to fit in memory so we need to store them partially or fully on disk. This is because secondary storage (on disk) is much cheaper than primary storage (RAM) These days, with the advent of Big Data, this type of scenario is a lot more common.

Here are some things to consider when we start using disk drives to store data:

- Disk drives are much slower than primary memory (RAM and Cache). See the example memory-vs-disk in your examples repository to run a quick experiment to compare the speeds.
- The slower speed leads us to amortize the cost of accessing the disk drive for reading and writing. By amortize, we mean to spread the cost over multiple operations so instead of reading one element at a time we read/write a block's (or a disk page) worth of elements in one operation. This works because a disk drive takes the same time to read one element or a block/page's worth of elements.
- **Implication for algorithm design**: Because disk access is often slower than processing all the information that was read, it makes sense to minimize the *number of disk operations*

along with the *CPU time*. So we will analyze of algorithms dealing with external storage for both parameters.

- **How do we store a data structure in a file on disk?**
  - **Solution 1**. Use a standardized format such as JSON (Javascript Object Notation). A human readable language independent text format consisting of attribute-value pairs and arrays. For example, given an instance of a class `Student`, we would store it in a JSON representation as shown below.

```
public class Student {
private int firstName;
private int lastName;
private long id;
}

----JSON representation of object returned by----
----new Student("Bronco", "Buster", 123456789)----

{ "firstName": "Bronco"
  "lastName": "Buster",
  "id": 123456789
}
```

    This is obviously easy to use, easy to pass objects between languages but it does have some downsides.

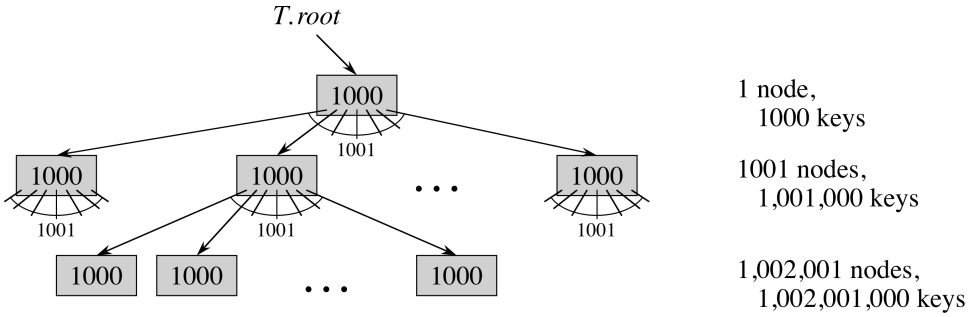
    * Advantage: Relatively easy to use, easy to pass to other languages.
    * Disadvantage:
      · Takes more space, especially for large collection of objects.
      · Take more processing time due to parsing.
  - Let's compare data stored in text files versus binary files. See examples in the repository folder disk-IO-examples. In particular, run and compare GenerateText.java and GenerateBinarySlower.java, and GenerateBinary.java, We will observe that binary files take less space and are faster to process. That leads us to the next solution for storing data structures on disk.
  - **Solution 2**. Use built-in serialization in Java (or other languages). Serialization is the act of saving a data structure as an array of bytes (that can be saved in a binary file). In Java, we can mark any class as being serializable by adding `implements Serializable` interface that has no methods. See the example StudentRecord.java, SaveTable.java, and LoadTable.java in the `disk-IO-examples` folder.

    * Advantage: Easy to use and compact.
    * Disadvantage:

        - The serialized output does not support random access so it would be very slow for use in a database or anywhere where we want to read selected objects out of the file (when files are too big to read all the objects into memory).
        - Not easy to pass to programs in other languages.
    - **Solution 3**. Build our own custom binary format for serializing objects. See the example external-binary-search that shows how we store an array of objects in a binary file that allows us to randomly access any object.
        * Advantage: Much faster for random access, compact.
        * Disadvantage:
            - Custom format is harder to deal with in our code.
            - Not easy to pass to other programs or languages.

Notes on B-Tree and disk access

- A **B-tree** is an example of solution 3. A B-tree will typically have a large branching factor so that each node fills up one disk page/block. Typical branching factors range from 50-2000, depending on the disk block size. A B-tree with a branching factor of 1000 can store a billion elements in a tree of height 2!



- B-tree algorithms copy selected pages from disk into the main memory as needed and write back onto the disk the pages that have changed.
- B-tree algorithms keep a constant number of pages in memory at any time. Thus, the size of the main memory does not limit the size of the B-tree that can be handled.
- Typical interaction with disk operations

```
Let x be a pointer to some object/node
DISK-READ(x)
Perform operations that access/modify attributes of x
DISK-WRITE(x)
We can still perform operations that access but not modify x
```

Definition of a B-Tree

A B-tree is a rooted tree (whose root is T.root) having the following properties.

1. Every node x has:
   (a) $x.n$, the number of keys currently stored in x
   (b) $x.n$ keys: $x.key_1 \leq x.key_2 \leq \ldots \leq x.key_n$
   (c) $x.leaf$, a Boolean which is true if x is a leaf and otherwise false
2. Each internal node x also contains $x.n+1$ points $x.c_1, x.c_2, \ldots, x.c_n, x.c_{n+1}$ to its children. Leaf nodes have no children so their c_i values are null or undefined.
3. The keys $x.key_i$ separate the ranges of keys stored in each subtree. If k_i is any key stored in the subtree with root $x.c_i$, then

$$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \ldots \leq x.key_n \leq k_{n+1}$$

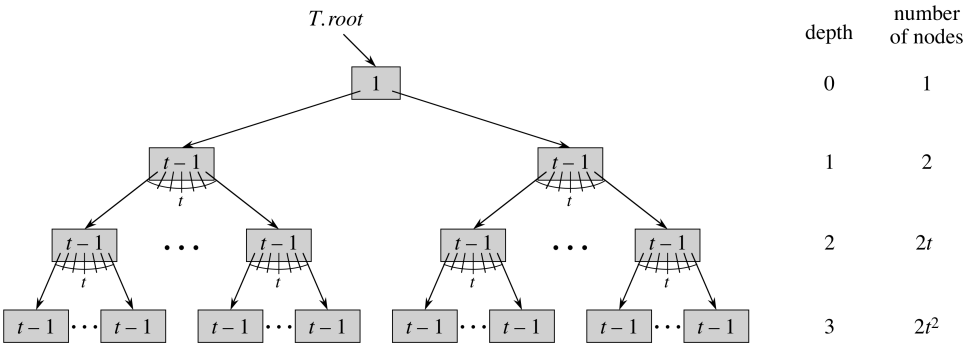
4. All leafs have the same depth, which is the height h of the tree.
5. The value $t \geq 2$, is called the minimum degree of the B-tree, helps define the structure of the tree as follows:
   (a) Every note other than the root must have at least $t-1$ keys. Every internal node other than the root has at least t children. If the tree is non-empty, the root mist have at least one key.
   (b) Every node may contain at most $2t-1$ keys. Thus an internal node may have at most $2t$ children. We say that a node is full if it contains exactly $2t-1$ keys.

Notes:

- The simplest B-tree is with minimum degree $t = 2$. That implies that each node may have 2, 3, or 4 children and we call such a tree a **2-3-4 tree**.
- If $n \geq 1$, then for any n-key B-tree of height h and min degree $t \geq 2$, we can say that

$$h \leq log_t \frac{n+1}{2}$$

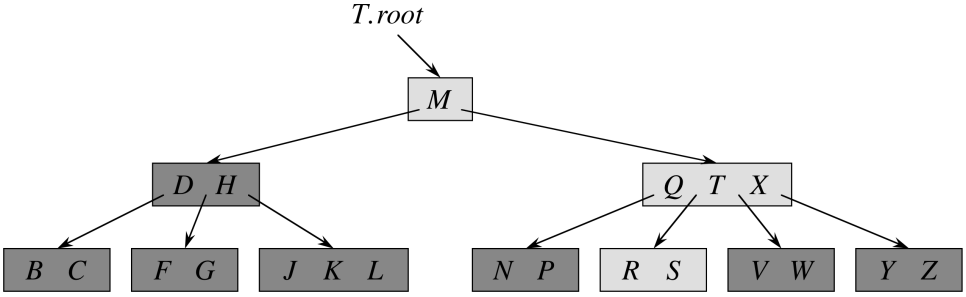
We can prove the above statement by drawing the tree counting the number of nodes as a function the height h.



$$
\begin{aligned}
n &\geq 1 + (t-1)\sum_{i=1}^{h}(2t^{i-1}) \\
&= 1 + 2(t-1)\sum_{i=1}^{h} t^{i-1} \\
&= 1 + 2(t-1)\frac{t^h - 1}{t-1} \\
&= 1 + 2t^h - 2 = 2t^h - 1 \\
n &\geq 2t^h - 1 \\
n+1 &\geq 2t^h \\
2t^h &\leq n+1 \\
t^h &\leq \frac{n+1}{2} \\
h &\leq \log_t \frac{n+1}{2}
\end{aligned}
$$

Recommended Exercises: Exercise 18.1-1, 18.1-2, 18.1-3, 18.1.4 (challenging!)

Exercise 18.1-2: Find all valid values of t for which the following is a valid B-tree.



Basic Operations

Assumptions:

- The root of the B-tree is always in main memory, so we never need to perform a DISK-READ on the root but we do need to perform a DISK-WRITE when the root node is changed.
- Any nodes that are passed as parameters already have had a DISK-READ operation performed on them.
- All procedures are "one-pass" algorithms that proceed downward from the root, without having to back up.
- We will access the key values and the child pointers using arrays inside the node x. They would need to be declared so they can hold a full node, so $2t-1$ keys and $2t$ child pointers but we will access only the following values:

  $x.key[1] \ldots x.key[x.n]$ and $x.c[1] \ldots x.c[x.n+1]$

Search

- Generalization of binary tree search except at node x we make an $(x.n+1)$-way branching decision.
- The return value is a 2-tuple (y,i) consisting of a node y and an index i such that $y.key_i = k$, where we are searching for key k. If not found, the search returns NIL.

```
B-TREE-SEARCH(x, k)
1. i = 1
2. while i <= n and k > x.key[i]
3.       i = i + 1
4. if i <= x.n and k == x.key[i]
5.       return (x, i)
6. elseif x.leaf
7.       return NIL
8. else  DISK-READ(x.c[i])
9.       return B-TREE-TREE-SEARCH(x.c[i], k)
```

$O(h) = O(\lg n)$ disk operations

$O(th) = O(t \lg n)$ run-time

Creating an empty B-tree

```
B-TREE-CREATE(T)
1. x = ALLOCATE-NODE()
2. x.leaf = TRUE
3. x.n = 0;
4. DISK-WRITE(x)
5. T.root = x
```

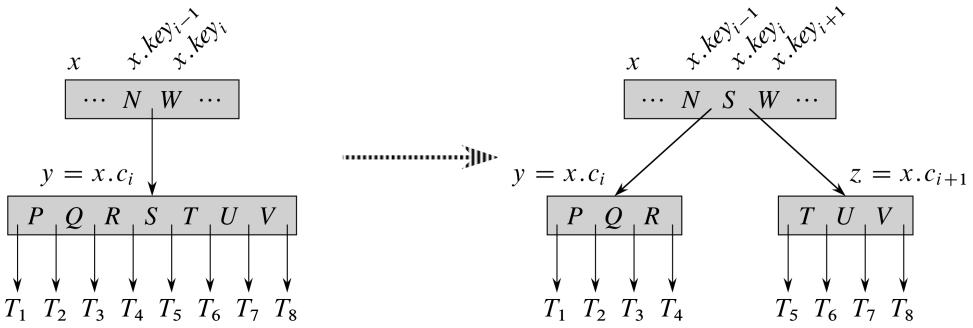
$O(1)$ disk operations and $O(1)$ CPU time.

Inserting a key into a B-tree

As we travel down the tree searching for the position where the new key belongs, we split each full node we come to along the way (including the leaf itself). Thus whenever we need to split a full node y, we are assured that its parent is not full.

Splitting a full node

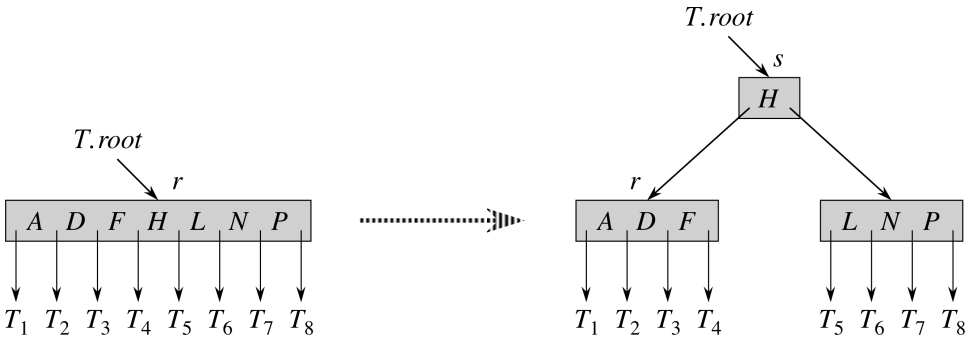
- Input is a non-full internal node x (assumed to be in the main memory) and an index i such that $x.c_i$ (also assumed to be in the main memory) is a full child node.
- The procedure then splits this child into two nodes ad adjusts x such that it has an additional child.
- To split a full root, we will first make the root a child of a new empty root node – thus the tree grows in height by one. Splitting the root is only means by which the B-tree grows.




```
B-TREE-SPLIT-CHILD(x, i)
1.  z = ALLOCATE-NODE()
2.  y = x.c[i]
3.  z.leaf = y.leaf
4.  z.n = t - 1
5.  for j = 1 to t-1
6.     z.key[j] = y.key[j+t]
7.  if not y.leaf
8.     for j = 1 to t
9.        z.c[j] = y.c[j+t]
10. y.n = t - 1
11. for j = x.n + 1 downto i+1
12.    x.c[j+1] = x.c[j]
13. x.c[i+1] = z
14. for j = x.n + 1 downto i
15.    x.key[j+1] = x.key[j]
16. x.key[i] = y.key[t]
17. x.n = x.n + 1
18. DISK-WRITE(y)
19. DISK-WRITE(z)
20. DISK-WRITE(x)
```

Inserting a key into a B-tree in a single downward pass

See below for an example of a root node getting split (for degree $t = 4$):

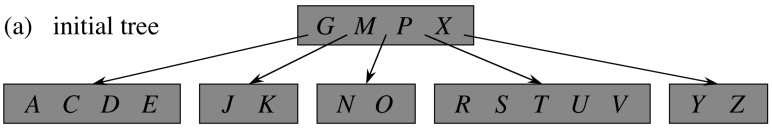


The root node R splits in two, and a new root node s is created. The new root node contains the median key of r and has two halves of r as children. The tree grows in height by one as a result.

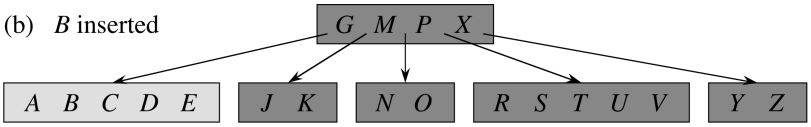
```
B-TREE-INSERT(T, k)
1.  r = T.root
2.  if r.n == 2t-1
3.       s = ALLOCATE-NODE()
4.       T.root = s
5.       s.leaf = FALSE
6.       s.n = 0
7.       s.c[1] = r
8.       B-TREE-SPLIT-CHILD(s, 1)
9.       B-TREE-INSERT-NONFULL(s, k)
10. else B-TREE-INSERT-NONFULL(r, k)
```

See below for examples of insertion for B-Tree with degree $t = 3$.

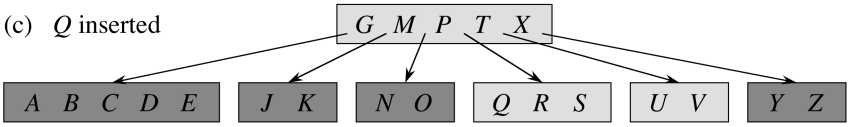
(a) initial tree



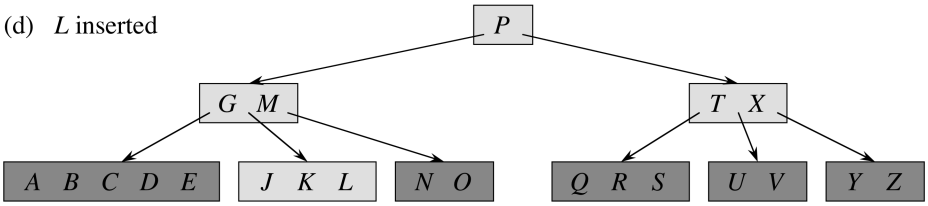
(b) *B* inserted



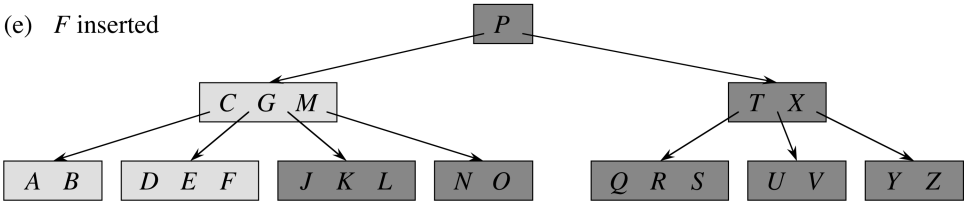
(c) *Q* inserted



(d) *L* inserted



(e) *F* inserted

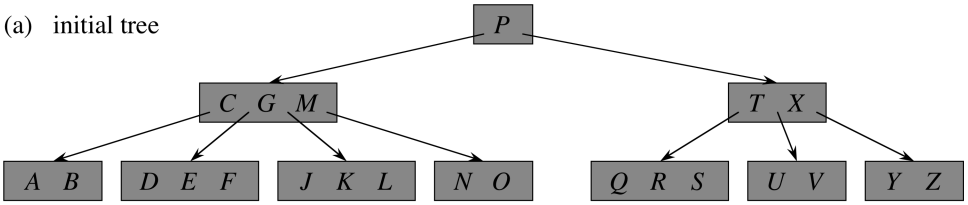


```
B-TREE-INSERT-NONFULL(x, k)
1.  i = x.n
2.  if x.leaf
3.      while i >= 1 and k < x.key[i]
4.          x.key[i+1] = x.key[i]
5.          i = i + 1
6.      x.key[i+1] = k
7.      x.n = x.n + 1
8.      DISK-WRITE(x)
9.  else while i >= 1 and k < x.key[i]
10.         i = i + 1
11.      i = i + 1
12.      DISK-READ(x.c[i])
13.      if x.c[i].n = 2t - 1
14.          B-TREE-SPLIT-CHILD(x, i)
15.          if k > x.key[i]
16.              i = i + 1
17.      B-TREE-INSERT-NONFULL(x.c[i], k)
```

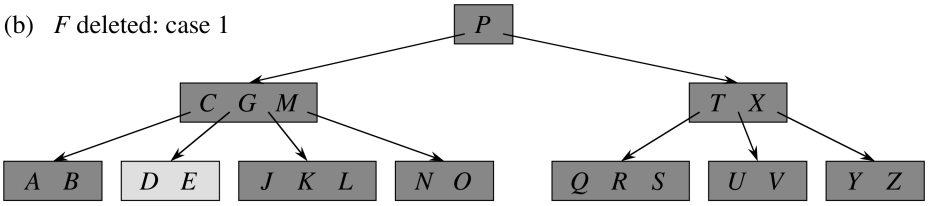
Recommended Exercises: Exercise 18.2-1, 18.2-2, 18.2-3.

Deletion

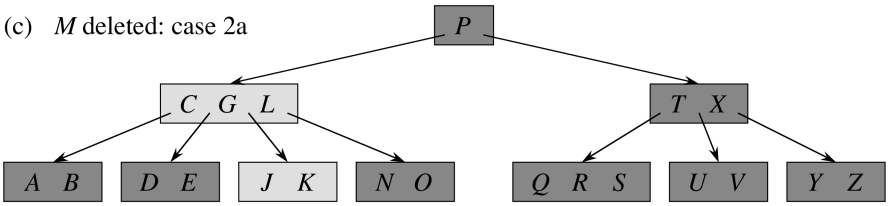
(a) initial tree



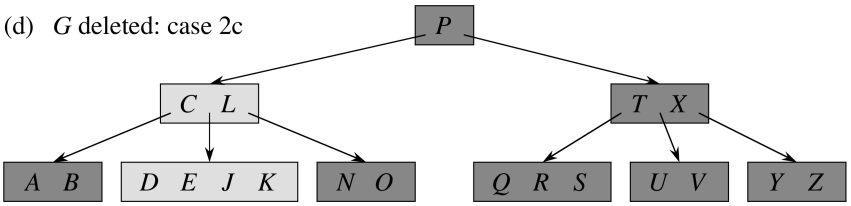
(b) *F* deleted: case 1



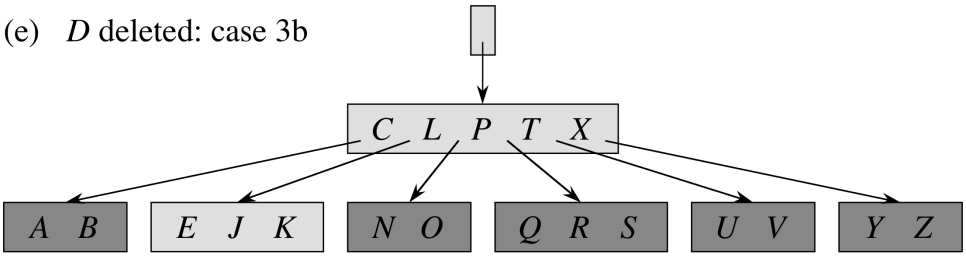
(c) *M* deleted: case 2a



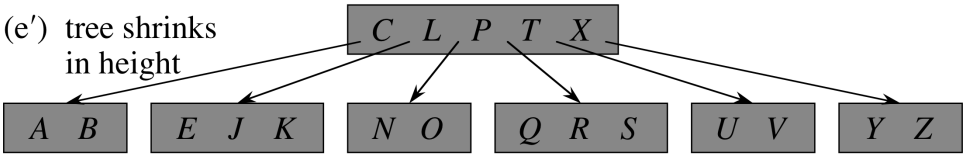
(d) *G* deleted: case 2c



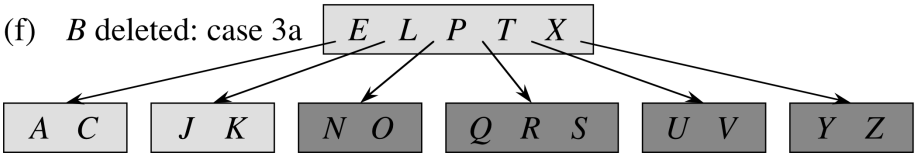
(e) *D* deleted: case 3b



(e′) tree shrinks in height



(f) *B* deleted: case 3a



Recommended Exercises: Exercise 18.3-1.

References

- BTree visualization: https://www.cs.usfca.edu/~galles/visualization/BTree.html