1. For the B+-Tree shown above (with at most 4 pointers per node), show the effect of following operations, in sequence:

(a) insert 80 
(b) insert 250 
(c) insert 280 
(d) delete 550
2. Suppose that we are using extendable hashing on a file that contains records with the following search-key values:

\[3, 8, 11, 14, 16, 17, 19, 20, 33, 43, 48\]

Show the extendable hash structure for this file if the hash function is\( h(x) = x \mod 7\), and buckets can hold 3 records each.

**ANSWER:**

Let’s write the hash values in binary first:

- \(H(3) = 3 = 00011\)
- \(H(8) = 1 = 00001\)
- \(H(11) = 4 = 00100\)
- \(H(14) = 0 = 00000\)
- \(H(15) = 1 = 00001\)
- \(H(16) = 2 = 00010\)
- \(H(17) = 3 = 00011\)
- \(H(19) = 5 = 00101\)
- \(H(20) = 6 = 00110\)
- \(H(33) = 5 = 00101\)
- \(H(43) = 1 = 00001\)
- \(H(48) = 6 = 00110\)

We will insert these in order (the order doesn’t really matter for the final result).

In the beginning, we just use the last bit (we can also use the prefix instead, but here it is easier to use the suffix). After inserting 3, 8, 11, 14, 15, 16, we still only have two buckets: \(0 = (11, 14, 16)\), and \(1 = (3, 8, 15)\).

Inserting 17 overflows the second buckets, and we start using two bits:

19 is inserted into the 2\(^{nd}\) bucket, but 20 causes us to split the first bucket into two (but we don’t need to increase the size of the suffix). 11 and 14 will go in one bucket, and 16 and 20 in the other.

Continuing, we need to split again and use another bit when inserting 33. Right one shows the final structure:
3. The hash join algorithm described in the textbook computes the natural join of two relations. Write pseudocode for an iterator that implements a hash-based algorithm for computing an “anti-join” operation: \( R \) anti-join \( S \) contains all tuples in \( R \) that have no matching tuples in \( S \). Although anti-join is not an operation that can be specified directly in SQL, it is implemented in the query processor to support some types of queries. Your pseudocode must define the standard iterator functions \( \text{open()}, \text{next()}, \) and \( \text{close()} \), and must fetch the input tuples using iterator calls (in other words, \( R \) and \( S \) may not be base relations, but may be outputs of other operators). Show what state information the iterator must maintain between calls. For simplicity, you can assume that \( S \) is small and easily fits in memory.

Here are more details on anti-join: [http://en.wikipedia.org/wiki/Relational_algebra#Antijoin](http://en.wikipedia.org/wiki/Relational_algebra#Antijoin)

**ANSWER:**

This is very similar to the hash join operator and so the answer is sketched here.

**Open():** Call \( \text{open()} \) on the children. Initialize the hash table.

**Next():**

First call:

- Read the entire left input (\( S \)) and build a hash table on it in memory.
- While( \( r = \text{right.next()} \) != null) {
  - Check if there is a match in the hash table.
    - If NO:
      - Return \( r \)
      /* If there is a match, then don’t return this, but read another tuple from \( R \) */
    }
  - Return null; // Exhausted \( R \)

**Close():** Call \( \text{close()} \) on the children; Delete Hashtable.

4. We would like to monitor the stock closing prices of companies as they change in time. We use the relation \( \text{Stock(co, date, val)} \) to store the closing value \( \text{(val)} \) of the stock of each company \( \text{(co)} \) at the end of each \( \text{date} \). This is a historic relation that at this time observes the following statistics:

- \( n(\text{Stock}) = 10,000,000 \) -- # tuples in the relation
- \( b(\text{Stock}) = 100,000 \) -- 100 tuples per page

(a) Estimate the I/O cost required to do a block-oriented nested-loop join with a 2-block buffer memory for the query:

\[
\text{select * from Stock S1, Stock S2 where S1.co = S2.co}
\]

(b) Do the same as in (a) with a 1001 block buffer.

(c) Estimate the I/O cost required to do a hash join for the query in (a).
ANSWER:

a) Since we have a self-join rather than a join of 2 distinct relations,
the I/O cost is $br \times br = 100000 \times 100000$ block retrievals.
The number of seeks is the same as if they were two different relations = $2br$.

b) The I/O will be as above $br \times br / 1000$.
The number of seeks will also be divided by 1000.

[Extra credit if anyone does this for 4a): a block $bi$ needs only to be joined with block $bj$ with $j \geq i$,
we only need to do $br + br - 1 + br - 2 + ... + br - (br - 1) = br \times br - (br - 1) / 2$.

The number of seeks in this case are only $br$ (1 to get to the 1st block + 1 to the 2nd on the second inner loop + 1 for the 3rd inner loop + ... + 1 for the last block).
]

c) Because it is a self-join, the retrievals will be $3 \times br$.
The seeks 1 to hash + $br$ to save and 1 to read the partitions again.