1 B+-tree Construction

Given a collection of N data records, and three alternatives to create a B+-tree on this data set:

1. Insert each record one at a time.
2. Sort data set, then perform record wise insertion.
3. Bulk loading.

What are the I/O costs and number of page accesses for each of the alternatives?

Suppose another set of m data records arrives. What are the costs (again, I/O and page accesses) for inserting the new data set into the existing tree. Consider the following three approaches:

1. Insert each record one at a time.
2. Sort data set, then perform record wise insertion.
3. Sort data set, merge with previous data and create a new tree using bulk loading.

Solution Sketch - needs review

Assuming the following cost model:

- Number of records per page: R
- Number of pages needed for storing leaf nodes: B (if pages are fully occupied B = N/R).
- Number of pages needed for storing intermediate nodes: E
- Fan-out: F
- Random Access cost: RA
- Sequential Access cost : SA

For creating a B+-tree from a collection of N data records:

1. Insert each record one at a time: For each record we need to search for its position on the tree then write it to the page. Searching involves fetching all pages from the root to the leaf plus a binary search on the leaf to find the correct position. Assuming splits at leaf node level are rare the number of page accesses is equal to $N \times \log_F B$. If all pages are on disk then there is one random access for each page accessed, i.e., I/O cost is $RA \times N \times \log_F B$. However, intermediate nodes are likely to be in the cache, so I/O cost would be $N \times RA$.
2. Sort data set, then perform record wise insertion: The number of pages accesses is the same as in the previous case ($N \times \log_F B$). Sorting the records improves I/O since access to leaf nodes can now be done sequentially, i.e, I/O = $N \times SA$. The I/O cost for external sorting is $O(B \log_M B)$, where M is the number of pages that fits into main memory. If the whole set fits into memory, then no external sorting is needed and I/O would be zero.
3. Bulk loading. We first need to sort the records, then each page has to be accessed only once, i.e. number of page accesses = $B + E$. I/O cost = $B \times SA$, assuming intermediate nodes in cache. Again, we also need to consider the cost for sorting: ($O(B \log_M B)$).

For inserting another set of m records:

1. Insert each record one at a time: Page accesses $= m \times \log_F b$ and I/O $= m \times RA$, where b is the number of pages needed for storing leaf nodes (see explanation for the case of N records).
2. Sorting is expected to improve (sequential with jumps) not sure how to quantify though. It seems that if the number of entries to be added is small, using the previous approach (inserting one record at the time) could be more efficient.
3. It involves sorting the m entries, merging the two data sets (remember that the bigger data set is already sorted), and performing bulk loading that costs $(B+b)+E'$ page accesses and $(B+b) \times SA$ I/Os (E' is the number of pages needed for storing intermediate nodes). Sorting and merging cost: $O(B \log_M B)$, if the data does not fit in main memory.

2 Hash indexes

Consider the Extendible Hashing index shown in Figure 1. Answer the following questions about this index:

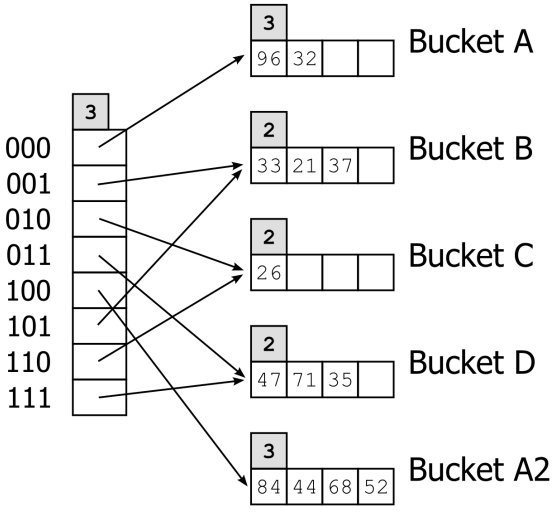

Figure 1: Extendible Hashing index.

1. Show the index after inserting an entry with hash value 20.
2. Show the index after inserting entries with hash values 49 and 85 into the original index.
3. Show the index after deleting the entry with hash value 26 into the original tree.

Solution Sketch - needs review

1. $20_{(10)} = 10100_{(2)}$. The value is inserted in bucket A2 which causes the bucket to split and the directory to double. See Figure 2.

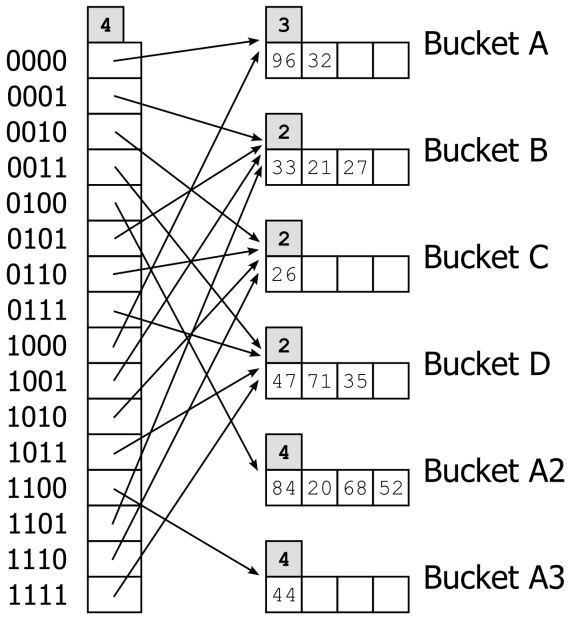

Figure 2: Solution 1.

2. $49_{(10)} = 110001_{(2)}$. $85_{(10)} = 1010101_{(2)}$. Both values are inserted in bucket B which causes the bucket to split. See Figure 3.

3. $26_{(10)} = 11010_{(2)}$. Bucket C remains empty after the deletion. It is not merged because its depth is 2 (one less than the global depth). See Figure 4.

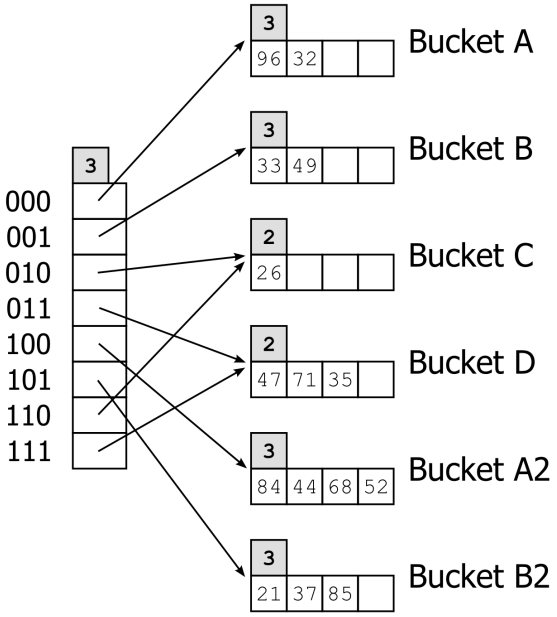

Figure 3: Solution 2.

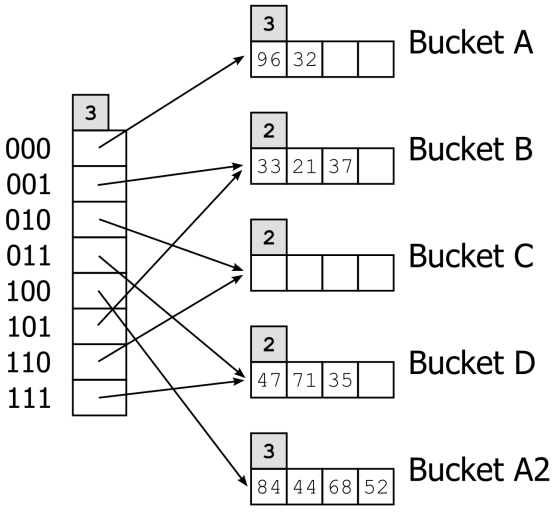

Figure 4: Solution 3.