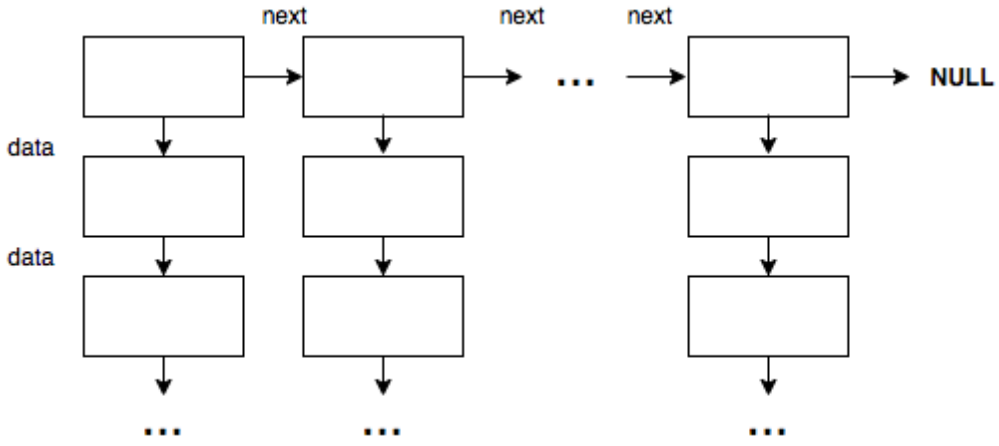
Step 3 asks to use a B+Tree data structure for indexing. The designed data structure in C resembles a B+Tree. It consists of a set of nodes connected to the next one, where each node is an element of the index, thus simulating the index. Each index node also points to an additional node containing additional information, for example, its leaves. The image below exemplifies our designed data structure. The leaves contain integers that represent ids of .dat files. The index is created and then saved as a .dat file. The B+ tree was created in a bottom-up fashion with an assumption that we already store the exiting records in the leaf nodes. Fan-out has been implemented as well and it should be inputted as command line argument. The indexes are generated in average 1.2 seconds. Building the indexes with different fan-outs does not impact the time create the indexes files. It takes in average, 1.2 seconds.



The four queries have been conducted and the results are shown in the table below.

	Fan-out 10	Fan-out 200
Q1	0.000005	0.00001
Q2	0.000753	0.000684
Q3	0.011744	0.01133
Q4	0.009972	0.011342

Comparing with the results from Stage 2, the indexing improves the time spend while querying for all the four queries and different fan-out. The discovery is that applying indexes techniques such as B+Tree, improves queries because the

operation will not need to go through all *n* values in a table/list. The queries will be performed within a range, based on the range of the index., not searching for all the elements in the data structure. Indexes can optimize specially when *n* becomes larger. It will happen because with larger *n*'s, the data can be divided in more blocks. Therefore, it is cheaper to check if data is within the range of a bucket and checking it all content. When *n* is small, the whole data can be compressed in only one bucket, which does not generate optimized queries because there will be no indexes.