CS 225: Data Structures

Assignment 8

Group D1

Last Modified on April 22, 2022

Members	
Name	**Student ID**
Li Rong	3200110523
Zhong Tiantian	3200110643
Zhou Ruidi	3200111303
Jiang Wenhan	3200111016

Please turn over for our answers.

Ex. 1 **Our Answer.** (This part is written by Zhou Ruidi)

We just need to compare the numbers (keys) in one node and find the point between two numbers and go to the next level. We will repeat this step until we find the "NULL" node which is in the last level before the leaf nodes (they are also empty and there is no point to these nodes). Then we will create a new pointer to the leaf node and use the leaf node space. (In short, we should "new" a pointer to find the location of the leaf node and then we can use the space). "insert" and "delete" operation should find the corresponding location at first and then we can "insert" or "delete". We can use this approach to reach the goal of handling the variation in the question.

Ex. 2 **Our Answer.** (This part is written by Zhong Tiantian)
Generally speaking, I/O operations only happen when a new sub-tree is loaded from the external disk into the memory.

(i) Assume each page operation costs $O(1)$.

Insert. The algorithm will try to find the correct place in a single pass down, at each "layer" I/O will cost $O(1)$ operation. In total, `insert` requires $O(H)$ page operations.

Oviously the height of a tree is at most

$$H = \log_t N.$$

Therefore, the total number of page operations should be

$$H \cdot O(1) = O(\log_t N).$$

Find. For B and B+ trees, the algorithm compares the given index with all index in a subtree, before which an I/O operation will happen, which costs $O(1)$. Such comparing process will continue until reaching the leaf nodes. Oviously the height of a tree is at most

$$H = \log_t N.$$

Therefore, the total number of page operations should be

$$H \cdot O(1) = O(\log_t N).$$

Delete. First the algorithm will try to locate the block of data to delete, which will cost at most $O(\log_t N)$ page operations as is mentioned above.

For a simple delete, just remove the block and this costs $O(1)$ page operation.

Although in other cases it costs more operations, they can be amortised into simple delete case. Therefore, the number of page operations is still

$$O(H) = O(\log_t N).$$

(ii) The algorithm will try to search all way with the left-most subtrees and find the lower bound; and with right-most subtrees to find the upper bound. In the worst case, the search process will go through all "layers" of the tree thus requires $O(2 \log_t N)$ since we are searching for two boundaries.

(iii) Each page operation will put exactly 1 block into the buffer. Since the B/B+ tree operations require at most $O(\log_t N)$ page operations, which means each time it puts

$$1 \cdot O(\log_t N)$$

blocks into the buffer. Thus we have

$$O(\log_t N) \leq k$$

and the solution for t will be

$$t > \sqrt[k]{N}.$$

The actual value of t will be actually smaller, since when performing one operation we may require other certain operation to be running at the same time and thus cost more page operations.

Note that although `range` operation requires $O(2 \log_t N)$ page operations, it can be splitted into two independent operations `find_max` and `find_min`, each requiring $O(\log_t N)$ and buffer space can be disposed after each operatoin, thus we only need at most $O(\log_t N)$ operations.

Ex. 3 **Our Answer.** (This part is written by Jiang Wenhan)
Firstly we select a node as root node, then we start at this node.

(a) we mark this node as searched node, put into the set T which records all the connected components of the given graph, and select one of its unmarked neighbor node.

(b) If its neighbor node has no unmarked neighbor, we mark this node and repeat step 1 until all neighbors are marked.

(c) If the neighbor node we select has unmarked neighbor node, then we mark this node and repeat step 1, 2 and 3 at this node.

All nodes marked (i.e. in the set T) are connected components of the given starting node undirected graph.

The following algorithm will state the above process more clearly.

Algorithm 1: Depth-Frist Search

input : v, T
output: T with all vertices connected to v
Procedure `DepthFirstSearch`(v, T)
  $T \leftarrow T \cup \{v\}$
  v.visited $\leftarrow$ true
  foreach edge *connected to* vertex **do**
    $v' \leftarrow$ the vertex at the other end of edge
    if v'.visited = false **then**
      call `DepthFirstSearch`(v')
    end
  end
  return T

Ex. 4 **Our Answer.** (This part is written by Jiang Wenhan)

Firstly we select a node as root node, and pushback it to the queue. Then we mark the nodes in the queue as examined and pushback all unmarked neighbors of the nodes in the queue. Then we popfront all marked nodes in the queue and repeat this procedure until no unmarked neighbors can be found.

The following algorithm will state the above process more clearly.

Algorithm 2: Breadth-First Search

```
append starting_vertex to fifo_queue
T← T∪{starting_vertex}
starting_vertex.visited ←true
while fifo_queue ≠ ∅ do
    vertex ← the front element in fifo_queue
    vertex.visited ← true
    T ← T ∪ {vertex}
    pop fifo_queue's front element
    foreach edge connected with vertex do
        append vertex at the other end of edge (if any) to fifo_queue
    end
end
return T
```