Group51

Yijie Shen

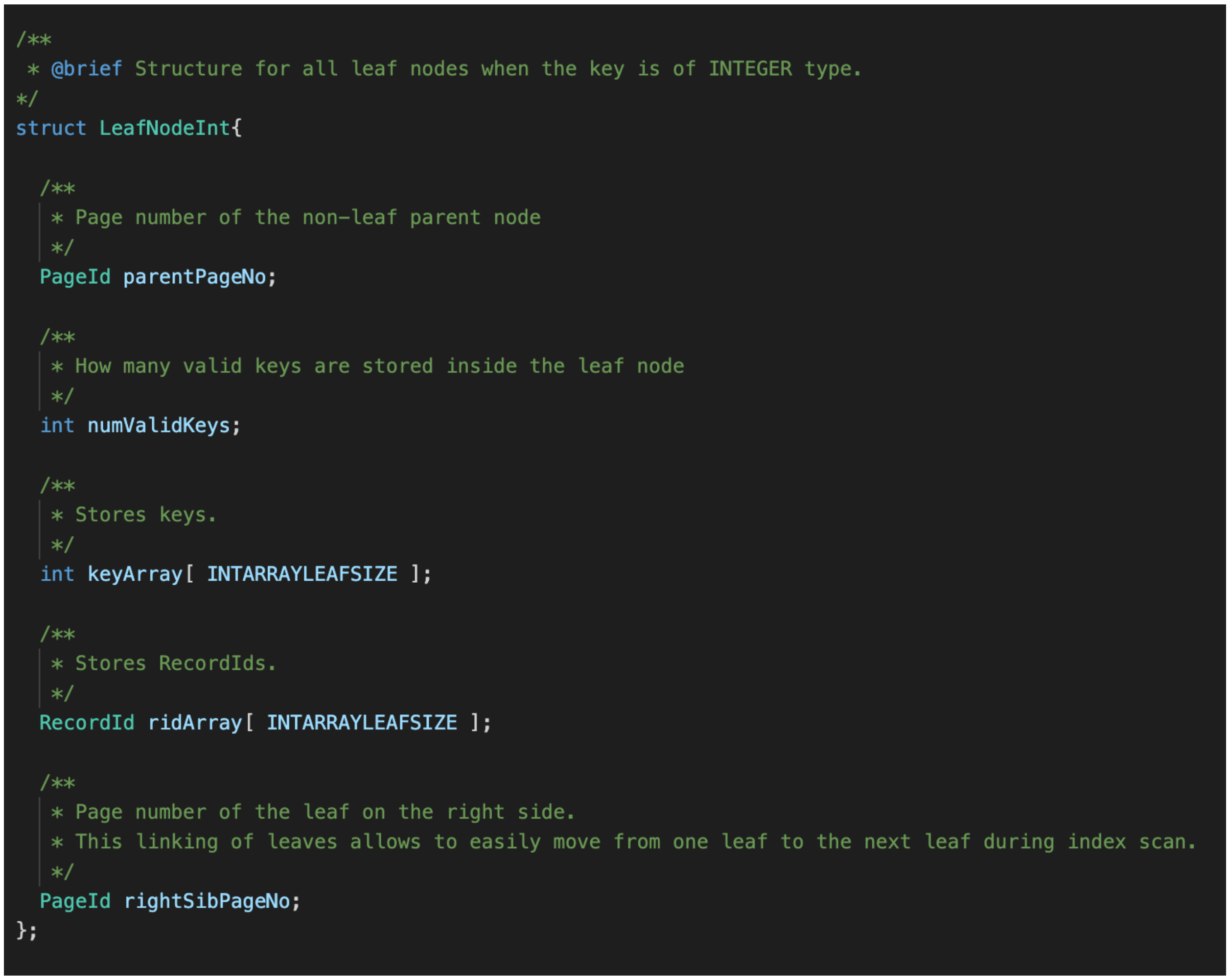
Liule Yang

Boya Zeng

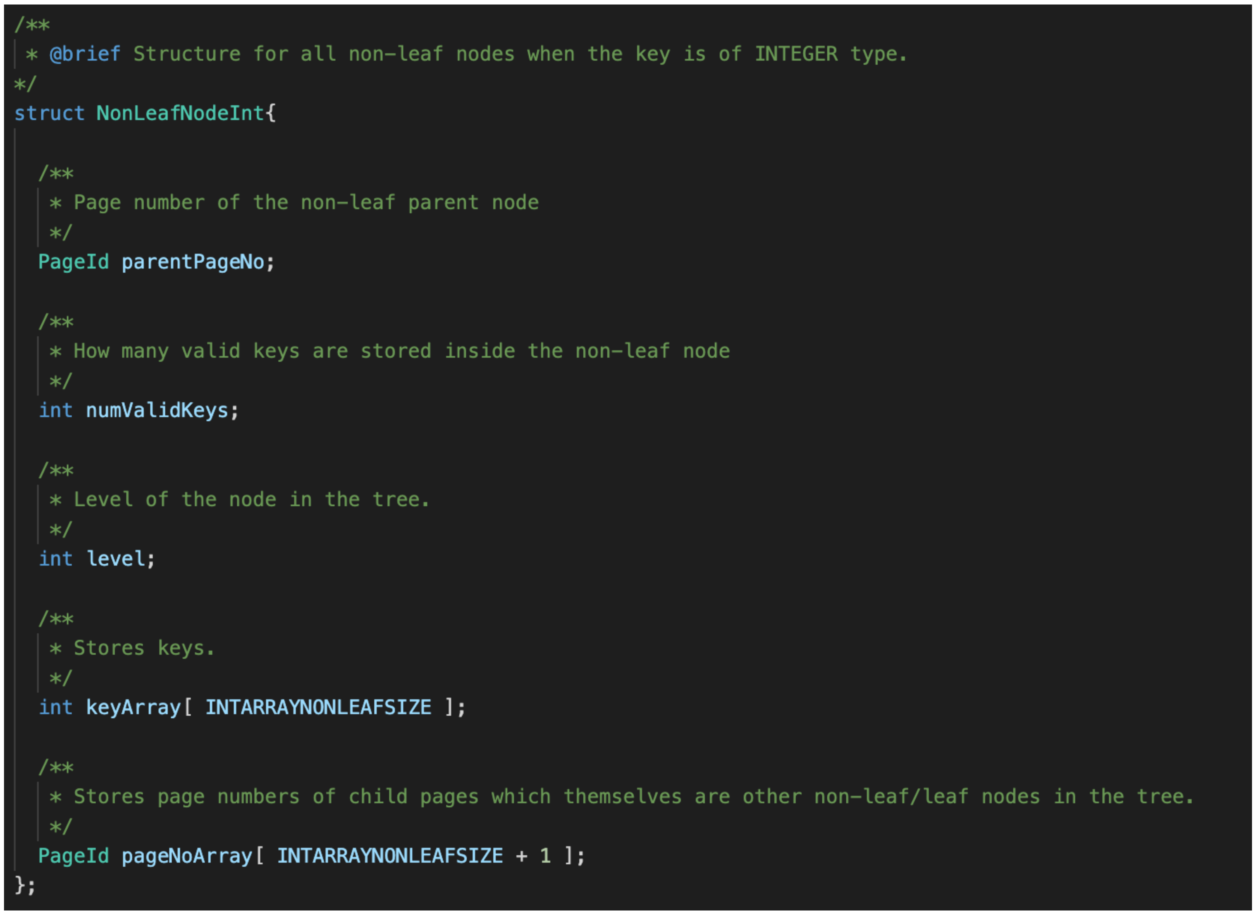
Design report

- **implementation choices & additional design choices & when to unpin**

Structure of two essential nodes:

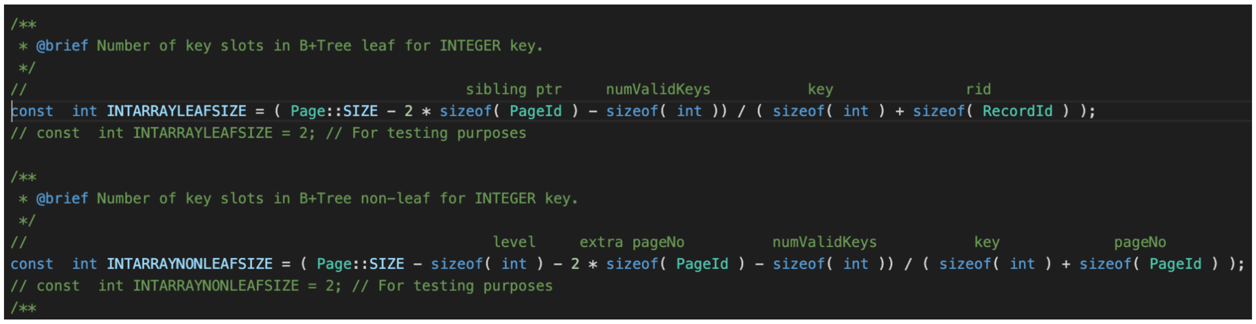


```
/**
 * @brief Structure for all leaf nodes when the key is of INTEGER type.
*/
struct LeafNodeInt{

  /**
   * Page number of the non-leaf parent node
   */
  PageId parentPageNo;

  /**
   * How many valid keys are stored inside the leaf node
   */
  int numValidKeys;

  /**
   * Stores keys.
   */
  int keyArray[ INTARRAYLEAFSIZE ];

  /**
   * Stores RecordIds.
   */
  RecordId ridArray[ INTARRAYLEAFSIZE ];

  /**
   * Page number of the leaf on the right side.
   * This linking of leaves allows to easily move from one leaf to the next leaf during index scan.
   */
  PageId rightSibPageNo;
};
```



```cpp
/**
 * @brief Structure for all non-leaf nodes when the key is of INTEGER type.
*/
struct NonLeafNodeInt{

  /**
   * Page number of the non-leaf parent node
   */
  PageId parentPageNo;

  /**
   * How many valid keys are stored inside the non-leaf node
   */
  int numValidKeys;

  /**
   * Level of the node in the tree.
   */
  int level;

  /**
   * Stores keys.
   */
  int keyArray[ INTARRAYNONLEAFSIZE ];

  /**
   * Stores page numbers of child pages which themselves are other non-leaf/leaf nodes in the tree.
   */
  PageId pageNoArray[ INTARRAYNONLEAFSIZE + 1 ];
};
```

(Changed the leaf/non-leaf array size accordingly)



```cpp
/**
 * @brief Number of key slots in B+Tree leaf for INTEGER key.
 */
//                                                  sibling ptr     numValidKeys           key               rid
const  int INTARRAYLEAFSIZE = ( Page::SIZE - 2 * sizeof( PageId ) - sizeof( int )) / ( sizeof( int ) + sizeof( RecordId ) );
// const  int INTARRAYLEAFSIZE = 2; // For testing purposes

/**
 * @brief Number of key slots in B+Tree non-leaf for INTEGER key.
 */
//                                                     level     extra pageNo          numValidKeys           key             pageNo
const  int INTARRAYNONLEAFSIZE = ( Page::SIZE - sizeof( int ) - 2 * sizeof( PageId ) - sizeof( int )) / ( sizeof( int ) + sizeof( PageId ) );
// const  int INTARRAYNONLEAFSIZE = 2; // For testing purposes
/**
```

We implement B+tree index on search keys of integer type by storing <key,rid> pairs to B+tree index.We use allocateLeafNode(leaf node) and allocateNonLeafNode (none leaf node) to represent two kinds of Node in the B+tree structure.

For insertion, we first initialize the leafNode (save one I/O per search, save disk space when relation is small and leaf page would be enough for the data entries)to insert, starting from the root node and traversing from top to bottom level by level(searchForLeaf). During the process, we unpin any pages traversed once we go to the next level. And finally the "searchForLeaf" method returns the PageId of the targetPageId of the leaf and unpin that page. Then the "insertToLeaf" method will do a bottom-up insertion from that targetPageId we just retrieved from the "searchForLeaf method". If the current node requires a split, a new page is allocated, information stored in the second half of the current node is cleared and moved to the newly created node, then pages for both nodes are unpinned with the dirty bit set to true. After insertion, the page for the current node is overwritten and unpinned with the dirty bit set to true. If the leaf node is full and needs to be split, the page for the newly created node is overwritten and unpinned with the dirty bit set to true simultaneously. Then when we are done with the current level, after unpinning any pages that have been changed, we go up to the parent level if there's a split happening. Then do the same insertion/split on the parent page over and over again until there's not a split any more. (If the root page has been changed, rootPageNum will change accordingly)

During the scanning process, we will find the page containing the given lower bound. If the current node is a non-leaf node, we get the next level's page number, unpin the page, and move to the next level's page number. If we reach page storing a leaf node, we will return the targetPageId with the page unpinned (dirty page set to false) (searchForLeaf method). Then in the startScan method we will read the page again, we will read the pinned leaf page until we have reached the last element stored in this page or the current entry is greater than the upper bound given. If we reach the end of this pinned leaf page, the next page pointed to by this page is pinned and the original page is unpinned. If we reach an element with a key greater than the upper bound, we finish the scan and the pinned page is unpinned. For end scan, if the currentPageNum != -1, we will unpin the page to make sure every page during scan was unpinned.

- **Efficiency of implementation**

For insertion, we first traverse from root to the leaf level to find the target leaf node PageId. Then we do a bottom-up insertion, and complete all the splits needed along the way by getting the parentPageNo of the current page.

For scan, we simply retrieve the leaf with the lower bound like we did in the insertion. And then just keep going right until reaching the end or the higher bound.

Worst case search time complexity: **O(logn)**

Average case search time complexity: **O(logn)**

Best case search time complexity: **O(logn)**

Worst case insertion time complexity: **O(logn)**

Average case Space complexity: **O(n)**

Worst case Space complexity: **O(n)**

- **Test**

Test1: Create a Forward relation with tuples valued 0 to relationSize and perform index tests

Test2: Create a Backward relation with tuples valued 0 to relationSize in reverse order and perform index tests

Test3: Create a Random relation with tuples valued 0 to relationSize in random order and perform index tests

Test4: Create a Random relation with relationSize = 50000 ,when leaf node would also split

Test5: Create a Random relation, test out_of_bound

Test6: Create a Random relation, test empty tree

Test7: Create Relation Forward Size with relationSize = 50, to test when there is only one leaf node for the entire tree (data number < 682)

Test8: Create Relation Forward Size with relationSize = 800 (split on leaf node only once)

Test9: Create Relation Forward Size with relationSize = 1200 (split on leaf node more than once)

Test10: First change the size of leaf node and non-leaf node both to 3 or 4, then Create Random Relation with relationSize = 30 to create a third level (non-leaf node split)

Test11: First change the size of leaf node and non-leaf node both to 2, then complete a specific sequence to test one case in the insertToNonLeaf function (index == midIndex++)

ErrorTests: test different error cases with regard to scan, to see if the correct exceptions are thrown at the right time.