CS6217 Project Report

Ivona Martinović and Guilhem Mathieux

November 2023

1 Introduction

B-trees and their variants, particularly B+ trees, are widely used for file organization. With applications extending from database management systems to file systems optimization, B+ trees play a pivotal role in minimizing input/output (I/O) operations and enhancing access methods across various domains [4, 5].

In the complex world of B+ trees, it's vital to keep the structure balanced and well-ordered. This is key to ensuring correctness and optimizing access times. A balanced structure is essential for preserving a shallow tree height, thereby guaranteeing predictable and efficient access. Formalizing the sorted order not only ensures correctness in isolation but it's also the foundation for a spectrum of complex algorithms—ranging from insertion and deletion to splitting, merging, and rebalancing.

The details of these algorithms, especially in the context of concurrent environments widespread in modern database systems, pose a significant challenge for formal verification. In such dynamic scenarios, where multiple clients concurrently access and modify data, ensuring the correct behavior of B+ trees becomes increasingly important.

Formal verification is a powerful approach that shows the correctness of a B+ tree implementation. By employing formal methods, we can specify preconditions, postconditions, and invariants in a clear and precise manner. Having these conditions written, makes it easier to understand the verification process.

In light of these considerations, our project was to create a formally verified implementation of a B+ tree in Dafny.

The report is structured as follows: Section 2 provides an overview of the definition of a B+ tree, while Section 3 delves into the details of our implementation. Subsection 3.1 explores the implementation of a the node class and its predicates, followed by Subsection 3.2, which outlines the higher level specifications used for the verification of B+ trees. Further, Subsection 3.3 defines the find method, and Subsection 3.4 introduces the insert method. The report concludes with Section 4, where we present our findings and discuss potential avenues for future work.

2 Description of B+ Trees

B+ tree is a specialized variant of the B-tree data structure characterized by distinctive organization of data. Much like the conventional B-trees, B+ trees are inherently balanced, ordered and exhibit n-ary tree properties. [1] However, the main difference is that B+ trees have have two types of nodes, internal nodes (also referred to as branch nodes) and leaf nodes.

Data in B+ trees is stored exclusively in leaf nodes. Each leaf node can store at most n pairs of key values and pointers to the storage location of the data matching the key, arranged in a sequential fashion. Additionally, each leaf node has an extra pointer referencing the next leaf node, which enables quicker range queries. Internal nodes consist of a sequence of at most n pairs of keys and $n+1$ subtrees. These key-subtree pairs are ordered in a way that values in a left subtree are less than the given key, while the values in the right subtree are greater or equal. [3, 5]

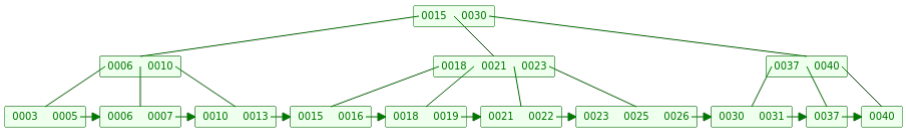

Figure 1: B+ tree visualization generated with [12]

Various papers use different definitions for the number of keys and children in a B+ tree, influencing what is known as the tree's order or branch parameter. In our work, we've chosen a specific definition for the order, denoted as n: it represents the maximum number of keys each node can hold. In this context, both internal and leaf nodes can have a maximum of n keys, and internal nodes can have at most $n+1$ children. Consequently, the minimum number of keys in any node (excluding the root) is $\lfloor n/2 \rfloor$. (As defined in [3])

Other papers propose alternative definitions, where the minimum number of keys in internal nodes is k and the maximum is then $2k$. Some distinctions also arise between internal and leaf nodes; for instance, certain papers state that leaf nodes have n entries, while internal nodes have $n-1$ keys or separators and n children (as seen in [6]). This introduces an alternative definition of order, focusing on the maximum or minimum number of children in a node. For example, in [11], a tree of order k is defined to have at least $k+1$ subtrees and at most $2k+1$.

Figure 1 illustrates a B+ tree with an order of 3. The tree exhibits balance, ensuring that the path from the root to any leaf has a consistent height. It is also ordered, adhering to defined minimum and maximum number of keys and children in each node. Additionally, the tree maintains alignment, where keys are sorted in internal nodes, and values in the left subtree are less than the key, while those in the right subtree are greater or equal to the key. The diagram further highlights that all leaves are interconnected and encompass all values,

with values in internal nodes matching those in leaves. The values are sorted from left to right within the leaves.

3 Code overview and specification

To develop our mechanized proof for B+ trees, we used the Dafny extension for Visual Studio Code (VS Code). This extension allows for in code highlighting of errors and is easily downloadable from the extension marketplace. .NET and the Dafny executable are required to run the extension.

In keeping with our initial goal of having a general proof of B+ trees, the tree's order defining the maximum number of keys a node can hold is denoted by the $ORDER$ constant, making it easily modifiable.

We organized our code in two files, one for the node object `BPTNode.dfy` and another for the tree operations `BPTree.dfy`.

The code can be found in the public github repository at this link : BPTree-verif

3.1 BPTNode

A node of the B+ Tree is modeled as a class with the following variables:

- *keys*: an array of length $ORDER$ containing the keys
- *children*: an array of length $ORDER$+1 containing the children nodes
- $keyNum$: an variable counting the number of keys in the node
- *height*: an variable representing the height of the node (starting at 0 for the leaves and going up)
- $isLeaf$: a Boolean defining weather the node is a leaf
- $nextLeaf$: the pointer towards the next leaf (used only for leaf nodes)

Additionally, two ghost variables aid the verification process:

- *Repr*: a set allowing us to specify what can be modified by a given node (containing all keys arrays, children arrays and nodes of the subtree starting at this node)
- *Contents*: a set containing all the keys in the subtree starting at this node.

As discussed in section 2, B+ tree need to maintain a few properties. To convert these three properties (alignment, balance and order) to specification, we divided them in smaller predicates :

- **Alignment**
  - `Hierarchy`: all keys in a given subtree are bounded by surrounding keys in parent node (lower key $\leq$ subtree keys $<$ higher keys)
  - `Sorted`: keys are sorted from left to right in the keys array
- **Balance**
  - `ChildHeightEq`: all children of a node must have the same height.
  - `LeavesHeightEq`: a node has a height value of 0 if and only if it is a leaf.
- **Order**
  - `HalfFull`: all non root nodes contain at least $\lfloor ORDER/2 \rfloor$ keys
  - `KeyNumOk`: contains exactly keyNum keys
  - `ChildNum`: contains one more child than it has keys.

All these predicate can be found in the code with the same names. These are all combined into the `Valid` predicate (shown in figure 2), which is the general predicate that should hold for any node.

Some additional properties are asserted in the `Valid` predicate:

- the *Repr* set contains everything that can be accessed by the node
- the *Contents* set contains all subtree keys
- the *height* is never less than 0
- the lengths of *children* and *keys* arrays are correct
- all children nodes are also `Valid`

```
ghost predicate Valid()
    reads this, Repr
    decreases height
    ensures Valid() ==> this in Repr
{
    this in Repr &&
    height >= -1 &&
    LengthOk() &&
    KeysInRepr() &&
    KeysInContents() &&
    children in Repr &&
    ( keyNum == 0 ==> Empty()) &&
    KeyNumOK()&&
    Sorted()&&
    LeavesHeightEq() &&
    (isLeaf==false ==> (
        nextLeaf == null &&
        ChildrenInRepr() &&
        ChildNum() &&
        ChildHeightEq() &&
        Hierarchy() &&
        NonCyclical() &&
        ChildrenContentsDisjoint() &&
        (forall i: int :: 0 <= i < keyNum+1 ==> (
            children[i].keys in Repr &&
            children[i].children in Repr &&
            children[i].Valid() &&
            children[i].Contents <= Contents)
        ) &&
        (keyNum > 0 ==> (
            Contents == SumOfChildContents(children[0..keyNum+1]) &&
            (forall num: int :: (num in Contents ==>
                num in SumOfChildContents(children[0..keyNum+1])
            ))
        ))
    ))
}
```

Figure 2: `Valid` predicate for BPTNode

3.2 BPTree

The tree as a whole is defined from the root, as all following nodes will be in the subtree of root. BPTree is a class with a single variable : *root*. When the class is first created, *root* is null. It is only when a value is inserted that a node is created and used as a *root*.

Similarly to BPTNode, some Ghost variables aid the verification process:

- *Repr*: a set allowing us to specify what can be modified by the tree.
- *Contents*: a set containing all the keys in the tree.
- *LeavesList*: a sequence containing all the leaves of the tree (in order).

From this file we can have more general information about the whole tree which allow for these additional predicates.

- `RootValid` : the root is a Valid node (and so are all its children, by the defintion of Valid predicate in BPTNode class).

- `HalfKeys` : all non root nodes must contain at least $\lfloor ORDER/2 \rfloor$ keys. This predicate is defined in BPTNode but is only tested from BPTree as it doesn't need to hold for the root (only checked for all children of the root)
- `LeavesValid` : all leafs contains extra pointer towards the next leaf if there is one and all keys appear in a leaf node.

These are in turn combined in `Valid` predicate (illustrated in figure 3 much in same logic as for the BPTNode.

```
ghost predicate Valid()
    reads this, Repr, LeavesList
{
    this in Repr &&
    (root == null ==> Contents == {}) &&
    (root != null ==>
        root in Repr && root.Repr <= Repr &&
        !(this in root.Repr) &&
        root.Valid() &&
        Contents == root.Contents &&
        root is BPTNode &&
        HalfKeys()&&
        LeavesValid()
    )
}
```

Figure 3: `Valid` predicate for BPTree

3.3 Find method verification

The find or search method in a B+ tree is a fundamental operation that aims to locate a specific key within the tree. The primary objective is to determine whether a given key exists in the tree.

The search typically begins at the root of the B+ tree. If the key is present in the tree, the search process will guide us to the specific leaf node where the key is stored. During the search, keys are compared to determine the direction in which to traverse the tree. If the search key is less than a particular node's key, the search continues in the left subtree; if it's greater, the search moves to the right. The search operation involves a recursive descent through the tree. At each internal node, the search narrows down the possible locations of the key until it reaches a leaf node. Upon reaching a leaf node, a final comparison is made to check if the key is present in the node. If it is, the search is successful, and the method returns an indication of the key's presence. If the search key is not found in the leaf node, the method indicates that the key is not present in the B+ tree. This information can later be used in insert or delete method.

The pseudocode for the find method is presented in Algorithm 1. Notably, this is a find helper function, defined as a static method. The actual Find

Algorithm 1 Find method

```
procedure FIND(node, value)
    if node is empty then
        return false
    else if node is leaf then
        for key in node.keys do
            if key == value then
                return true
        return false
    else                                          ▷ node is an internal node
        for i in range(node.keyNum) do
            if value < node.keys[i] then
                return FIND(node.children[i], value)
        return FIND(node.children[node.keyNum], value)
```

method within the BPTree class receives only the target *value* as an input and subsequently invokes this helper function. In this invocation, the *node* parameter is set to the root of the B+ tree, and the *value* parameter is identical to the input argument.

```
method Find(val: int) returns (inTree: bool)
    requires Valid()
    ensures root == null ==> inTree == false
    ensures root != null ==> (root.ContainsVal(val) <==> inTree)
    ensures Valid()
{
    if root == null { return false; }
    else { inTree := FindHelper(root, val); }
}
```

Figure 4: Find method implementation and specifications

In Figure 4, the complete code for the Find method, including specifications, is presented. While this method is a part of the BPTree class, the primary logic resides in the FindHelper method. The crucial preconditions and postconditions ensure the initial and final validity of the B+ tree. The return value, denoted as *inTree*, is `false` if the tree is empty (indicating a null root). When the tree is not empty, the return value corresponds to the result of a ghost predicate, ContainsVal, defined in the BPTNode class. This predicate checks whether the sought-after value is present in the ghost variable *Contents*. As mentioned earlier, this variable encompasses all values in the subtree (or the entire tree, in the case of the root node).

Figure 5 displays the specifications for the FindHelper method. This static method takes a subtree, which is the current focus of the search, and the target value as input. It requires that the provided node, and its subtree, are valid, which is checked with Valid predicate from BPTNode class. The method ensures

```
method FindHelper(node: BPTNode, val: int) returns (inTree: bool)
    requires node.Valid()
    ensures node.ContainsVal(val) <==> inTree
    decreases node.Repr
```

Figure 5: Find helper specifications

that its return value aligns with the outcome of the ContainsVal predicate. Due to its recursive nature, Dafny necessitates a decreases clause to establish termination. In this context, we specify that the representation of the input node decreases, referring to a ghost variable encapsulating all nodes in the given subtree, including their keys and children arrays.

We successfully verified both functions (Find and FindHelper), although the verification process required the inclusion of numerous `assert` clauses. In the case of the FindHelper method, which consists of 66 lines of code (LOC), eliminating `assert` and `invariant` clauses results in 32 LOC, representing 48.5% of the original code. Asserts in Dafny are a way to guide the proof in the correct direction to help Dafny find a path towards full verification. Here they were particularly needed to prove that if a node is not in the subtree where it should have been inserted, then it is not anywhere else.

3.4 Insert method

The Insert method in a B+ tree serves the purpose of incorporating a new value into the tree while maintaining its structural integrity. This involves inserting the value into a suitable leaf node. If the leaf node is already at maximum capacity, a split operation is triggered. During a leaf split, a new key is generated, prompting an insertion into an internal node. If the internal node is also at full capacity, it necessitates a split. This sequence of operations may propagate upwards through the tree, potentially resulting in an increase in the overall height of the tree.

Algorithm 2 provides the pseudocode for the InsertHelper method, which operates as a helper method for the main Insert method defined in the BPTree class. It is employed within the primary Insert function of the BPTree, where only the *value* to be inserted is provided as an input. The helper method is then called with the *node* parameter set to the B+ tree's root, *hasParent* initialized as `false`, and the *value* parameter reflecting the input argument.

```
method Insert(val: int)
    requires Valid()
    modifies Repr
    ensures Valid()
    ensures val > 0 ==> Contents == old(Contents) + {val}
    ensures val <= 0 ==> Contents == old(Contents)
```

Figure 6: Insert specifications

Algorithm 2 Insert method

```
procedure INSERT(node, hasParent, value)
    if value already in the tree then
        return null, false
    else if node is null then                          ▷ edge case: B+ tree is empty
        create a new node with one key equal to the value
        return node, false
    else if node is leaf then
        if node.keyNum < ORDER then ▷ there is enough space in the leaf
            INSERTATLEAF(node, value)
            return node, false
        else                                                ▷ leaf needs to split
            splitNode = SPLITLEAF(node, value)
            if hasParent ==false then                       ▷ if the root was leaf
                create a newNode with one key equal to the first key in
                  splitNode and two children, node and splitNode
                return newNode, false
            else                          ▷ the parent node will have to be updated
                return splitNode, true
    else                                                ▷ node is an internal node
        index =GETINSERTINDEX(node, value)
        newNode, updateParent =
          INSERT(node.children[index], true, value)
        if updateParent then
            if node.keyNum < ORDER then          ▷ internal node is not full
                INSERTINNODE(node, newNode)
                return node, false
            else                                      ▷ internal node has to split
                newNode = SPLITNODE(node, newNode)
                return newNode, true
```

Figure 6 details the preconditions and postconditions for the insert method. This function necessitates the validity of the tree, modifies the *Repr* ghost variable containing all BPTNodes and their arrays, and ensures the tree remains valid after the insertion. Additionally, it guarantees that if the input argument value is non-positive, the tree remains unchanged, as indicated by the *Contents* ghost variable, a set encompassing all values in the tree. If the *value* is positive, it should be inserted, resulting in the *Contents* variable having one more element, namely the value being inserted.

Figure 7 shows specifications for InsertHelper method. This method is close to the pseudo code described in 2. It has two return variables:

- *newNode* which is either a new node that has been created in a split or the modified input node if no split was required,

- *updateParent* a boolean to show whether the parent needs to incorporate the *newNode* (true in case of split).

It needs to ensure that

- both nodes are Valid
- the value to be inserted is in one of the nodes
- newNode is either the previous node or a fresh node
- updateParent can only be true if there is a parent
- only the value to be inserted should have changed in contents
- if no split was needed, the newNode should contain the same keys as the previous node with the addition of x

```
static method InsertHelper(node:BPTNode, hasParent:bool, x:int) returns (newNode:BPTNode, updateParent:bool)
    requires node.Valid()
    requires x !in node.Contents
    requires x > 0
    modifies node, node.Repr
    ensures  newNode.Valid() && node.Valid()
    ensures  node.ContainsVal(x) || newNode.ContainsVal(x)
    ensures  newNode == node || fresh(newNode)
    ensures  !hasParent ==> !updateParent
    ensures  updateParent ==> node.Contents + newNode.Contents == old(node.Contents) + {x}
    ensures  !updateParent ==> newNode.Contents == old(node.Contents) + {x}
    decreases node.height
```

Figure 7: InsertHelper specifications

While attempting to verify the InsertHelper function in Dafny, we encountered challenges with the overall complexity and length of the function. Dafny struggled to reason about it within a reasonable time frame, even with a generous limit of 10 minutes. Recognizing the need to simplify the code, we opted to break it down into smaller, more manageable parts. The provided pseudocode outlines the names of these smaller functions, each designed to facilitate the verification process. Even with the use of those functions, the InsertHelper function only partially verifies. The first part of the function dealing with the node if it is a leaf has been independently verified but adding the next part results in time outs. The Insert also times out due to what we believe to be a problem of representation. It seems like changing the root pointer confused the compiler but we didn't have the time to look into this issue.

The smaller functions posed challenges, resulting in timeouts despite extending Dafny's Verification Time Limit to 300 seconds. The subsequent section will delve into a detailed discussion of these smaller methods, including their respective pre and postconditions.

As illustrated in Algorithm 2, the algorithm first checks for the presence of the value in the tree. If the value is already present, it signals that no further insertion is required. In the event that the tree is empty, the algorithm creates

a new node with the value as the key. When dealing with a leaf node that has space for the new value, it invokes the InsertAtLeaf method to perform the insertion.

The InsertAtLeaf method has been implemented in the BPTNode class and figure 8 displays the specifications for this method. It requires that the node into which the insertion occurs is a valid leaf node, not exceeding its full capacity (indicating that the key fits into the key array, or the number of keys is less than the maximum allowed). Additionally, it ensures that the value being inserted, denoted as *key*, is not already present in this node and is a valid value (greater than 0). This method modifies the node and its array, guaranteeing the node's validity after the insertion. It asserts the inclusion of the inserted value in the *Contents* ghost variable and ensures that the *Contents* after the execution is equal to the *Contents* before the execution with the added *value*.

In this method, it is notable that Dafny encounters considerable challenges when dealing with sets, a subject we explore in greater detail in Section 4. As a result of these challenges, the verified method contains four lines of code and 15 assert lines (3.75 times more). It is conceivable that not all of these assertions are essential, and there may be room for improving them. However, due to the extended verification time for this method and our current level of expertise, this represents the best approach we could devise.

```
method InsertAtLeaf(key:int)
    // insert a key value in a node
    requires Valid()
    requires isLeaf == true
    requires NotFull()
    requires !(key in Contents)
    requires key > 0
    modifies this, keys
    ensures Valid()
    ensures ContainsVal(key)
    ensures old(Contents) + {key} == Contents
```

Figure 8: InsertAtLeaf specifications

The key method within InsertAtLeaf is a crucial helper function called InsertIntoSorted, responsible for adding a new value to a sorted array. Specifications for this method are detailed in Figure 9. Given that these arrays represent our *keys* arrays, they are sorted up to a specific limit (*keyNum* in our case). Beyond this limit, from the *limit* index to the end, the values in the array are set to zero. Values before *limit* index are greater than zero, the *key* intended for insertion is not already present in the array, and it is greater than zero.

Upon completion, the method returns a new array b with the same length. This new array is also sorted, but the limit is now increased by 1 (to account for the inserted value). All values up to the new limit are greater than zero, and beyond $limit + 1$, the values are set to zero. The inserted *key* is present in the new array, and previous values from the input array a are retained in the newly created array.

The method initially appends the new value to the end of the sorted segment

```
method InsertIntoSorted(a: array<int>, limit:int, key:int) returns (b: array<int>)
    requires key > 0
    requires key !in a[..]
    requires 0 <= limit < a.Length
    requires forall i :: 0 <= i < limit ==> a[i] > 0
    requires forall i :: limit <= i < a.Length ==> a[i] == 0
    requires sorted(a[..limit])
    ensures b.Length == a.Length
    ensures sorted(b[..(limit+ 1)])
    ensures forall i :: limit + 1 <= i < b.Length ==> b[i] == 0
    ensures forall i :: 0 <= i < limit ==> a[i] in b[..]
    ensures forall i :: 0 <= i < limit + 1 ==> b[i] > 0
    ensures key in b[..(limit+1)]
    ensures forall k :: k in a[..limit] ==> k in b[..(limit+1)]
    ensures forall k :: k in b[..(limit+1)] ==> (k!=key ==> k in a[..limit])
```

Figure 9: InsertIntoSorted specifications

of the array, located at the *limit* index. Subsequently, it iteratively swaps this element with its predecessor if the preceding element is greater than the *key*. The process is encapsulated within a while loop, and when handling arrays in while loops, Dafny necessitates numerous invariants to facilitate reasoning about the array post-loop. The while loop is depicted in Figure 10. The figure illustrates that only array b is altered, reaffirmed by the $a[..] = a'$ invariant, where a' is a ghost variable representing the sequence into which the array a was initially copied at the beginning of the method. Predicates such as `lessThan` and `greaterThan` are employed to verify the sorted order of values in various sections of the arrays.

Following the loop, we incorporate assertions to assist Dafny in comprehending that array b encompasses a portion of a, the newly inserted value *key*, and the remainder of array a. Additionally, we include an assertion to verify the sorted order of array b. Subsequent to this, the code involves populating the remaining section of array b with zeros. Prior to this step, we aid Dafny in deducing that all elements of array a indeed exist in array b. Although the verification successfully confirms that b is constructed from a up to index k and a from index k to the end, Dafny requires additional assistance in reaching this conclusion independently. To address this, we introduce the helper lemma - DistributiveIn. Lemmas in Dafny are theorems used to prove another result, rather than being the goal itself. These lemmas help Dafny break the verification in smaller steps and can help greatly with verification process. Our DistributiveIn lemma is outlined in Figure 11. Another noteworthy observation is that despite asserting the inclusion of all elements up to index k in the second array and subsequently asserting the inclusion of all elements from index k in the second array, the final assertion affirming the inclusion of all elements from a in b could not be verified without the implication before this final assertion.

Following the examination of the scenario where the leaf has available space, the second helper function in the InsertHelper method is invoked when the

```
var i:= 0;
while (i < limit)
    modifies b
    invariant 0 <= k <= i <= limit
    invariant b.Length == a.Length
    invariant a[..] == a'
    invariant lessThan(a[..i], key) ==> i == k
    invariant lessThan(a[..k], key)
    invariant b[..k] == a[..k]
    invariant b[k] == key
    invariant k < i ==> b[k+1..i+1] == a[k..i]
    invariant k < i ==> greaterThan(b[k+1..i+1], key)
    invariant 0 <= k < b.Length && b[k] == key
{
    if(a[i]<key)
    {
        b[i]:= a[i];
        b[i+1] := key;
        k := i+1;
    }
    else if (a[i] >= key)
    {
        b[i+1] := a[i];
    }
    i := i+1;
}
assert b[..limit+1] == a[..k] + [key] + a[k..limit];
assert sorted(b[..limit+1]);
```

Figure 10: InsertIntoSorted while loop

```
lemma DistributiveIn(a: seq<int>, b:seq<int>, k:int, key:int)
    requires |a| + 1 == |b|
    requires 0 <= k <= |a|
    requires b == a[..k] + [key] + a[k..]
    ensures forall i :: 0 <= i < |a| ==> a[i] in b
{
    assert forall j :: 0 <= j < k ==> a[j] in b;
    assert forall j :: k <= j < |a| ==> a[j] in b;
    assert ((forall j :: 0 <= j < k ==> a[j] in b) && (forall j :: k <= j < |a| ==> a[j] in b))
                ==> (forall j :: 0 <= j < |a| ==> a[j] in b);
    assert forall j :: 0 <= j < |a| ==> a[j] in b;
}
```

Figure 11: DistributiveIn Lemma

leaf reaches its full capacity and necessitates a split, utilizing the SplitLeaf method. Specifications for the split leaf function are illustrated in Figure 12. Unfortunately, our attempts to verify this method were unsuccessful.

Nevertheless, let us delve into the specified preconditions and postconditions, integral to the verification of the InsertHelper method. The SplitLeaf method takes a node, where the value will be inserted, and the value itself as input. It requires that the node is valid, the value is not already present in this node, and

```
static method SplitLeaf(current:BPTNode, val:int) returns(newNode:BPTNode)
    modifies current, current.Repr
    requires current.Valid()
    requires !(val in current.Contents)
    requires val > 0
    requires current.isLeaf
    requires current.keyNum == ORDER
    ensures current.Valid()  && newNode.Valid()
    ensures current.isLeaf    && newNode.isLeaf
    ensures current.HalfFull() && newNode.HalfFull()
    ensures current.keyNum > 0 && newNode.keyNum > 0
    ensures current.keys[current.keyNum-1] < newNode.keys[0]
    ensures fresh(newNode)
    ensures old(current.Contents) < (current.Contents + newNode.Contents)
    ensures val in (current.Contents + newNode.Contents)
    ensures current.Contents !! newNode.Contents
    ensures forall k :: k in newNode.Contents ==> (newNode.keys[0] <= k)
    ensures forall k :: k in current.Contents ==> (k < newNode.keys[0])
```

Figure 12: SplitLeaf specifications

the value is a valid entity, greater than zero. Given that this method is employed to split a leaf, an additional prerequisite is that the provided node is a leaf and is fully occupied, denoted by its $keyNum$ equating the maximal number of keys in a node, represented by the constant $ORDER$. Upon completion, the method should instantiate another node, denoted as $newNode$. Both the initial node and the newly created node (verified with the fresh predicate) should be valid, serving as leaves with HalfFull predicates satisfied (ensuring the minimum number of keys). Although this inherently implies that $keyNum$ is greater than zero, subsequent ensure clauses are articulated to prevent Dafny from reporting an `index out of bounds` error when accessing index $keyNum - 1$ in the next ensure clause. This ensures clause guarantees that the last value in the current node is less than the first value in the newly created node. The ensure clauses with $Contents$ variables in it pose challenges for Dafny. First one says that the $Contents$ of the input node should be a subset of the combined $Contents$ of that node and the $newNode$, given the redistribution of keys. The inserted value should be in the union of $Contents$ variables, and these sets should be disjoint. Furthermore, all values in the first leaf should be less than the first key in $newNode$, as this value will become the key/separator in their parent node. Consequently, this first key in $newNode$ should also be less than or equal to all keys in the $newNode$.

Following the node split, we examine whether the input flag, $hasParent$, is set to `false`, signifying that the node lacks a parent (i.e., it is the root). In such a case, we invoke the CreateNewParent function. This function generates a new internal node with the initial key of the split node, establishing its children as the original leaf and the newly split leaf. The specifications for this function are detailed in Figure 13.

This method mandates that the nodes, for which a parent node is being

```
static method CreateNewParent(firstChild:BPTNode, secondChild:BPTNode) returns (newParent:BPTNode)
    requires firstChild.Valid() && !firstChild.Empty()
    requires secondChild.Valid() && !secondChild.Empty()
    requires firstChild.height == secondChild.height
    requires forall k :: k in firstChild.Contents ==> (k < secondChild.keys[0])
    requires forall k :: k in secondChild.Contents ==> (secondChild.keys[0] <= k)
    ensures newParent.Valid()
    ensures forall k :: k in firstChild.Contents ==> k in newParent.Contents
    ensures forall k :: k in secondChild.Contents ==> k in newParent.Contents
    ensures firstChild.Valid()
    ensures secondChild.Valid()
    ensures fresh(newParent)
```

Figure 13: CreateNewParent specifications

created, are both valid and non-empty. Additionally, they must share the same height, and the values in the first child should all be less than the first key in the second node. This key will become the key in the newly created parent. Similarly, all values in the second node must be greater than or equal to this first key in the second child. This condition is already ensured by the valid predicate, but explicitly stating it aids Dafny in its reasoning process. After the execution of this method, the newly created node, denoted as *newParent*, must be valid, and all values from both of its children must be present in its *Contents*. Furthermore, these children nodes must remain valid. Since this method only modifies the *newParent*, which is created within this method, no modifies clause is necessary.

In the case that the splitted node had a parent, then the algorithm will return the split node and true to signal that the parent needs to be updated (at least it needs to insert the new key, or even split). With that, we come to the case if the node is an internal node.

In this situation, we first determine the insertion index using GetInsertIndex and recursively call the InsertHelper function on the appropriate child. If the child's insertion necessitates updating the parent, the algorithm checks if there's space in the internal node. If space is available, the new node is inserted into the internal node using InsertInNode. The specification for this function is provided in Figure 15. Unfortunately, we were unable to verify this method due to the timeout it causes.

```
static method InsertInNode(node:BPTNode, newNode:BPTNode)
    requires node.Valid()
    requires newNode.Valid()
    requires node.NotFull()
    requires !(newNode in node.Repr)
    requires !node.ContainsVal(newNode.keys[0])
    modifies node, node.keys, node.children
    ensures node.Valid()
    ensures (newNode in node.Repr)
```

Figure 14: InsertInNode specifications

The InsertInNode method is analogous to the InsertAtLeaf method. However, in addition to inserting a value into the *keys* array, it also inserts the newly created node into the appropriate index in the *children* array. In this context, *node* represents the parent node into which we are inserting, and *newNode* is the child being inserted. This method requires that both nodes are valid, that the parent node is not full (indicating its *keyNum* is less than the maximum), that *newNode* is not already in the subtree of *node* (specified using the *Repr* ghost variable), and that the parent node does not already contain the key to be inserted (which is the first key from the child node). The method modifies the parent node, as well as its *keys* and *children* arrays. Following the execution of this method, the *node* should be valid, and *newNode* should be part of its subtree.

Similar to the case of leaves, if there is no space in the internal node for a new key, the algorithm splits the internal node using the SplitNode method. This helper method returns the newly created node and `true` to indicate that its parent needs to be updated. The specifications of this method are shown in figure 15. We did not succeed in verifying this function, which is expected given the higher complexity of this method compared to SplitLeaf, which we also did not verify.

```
static method SplitNode(current:BPTNode, child:BPTNode) returns(newNode:BPTNode)
    modifies current, current.Repr
    requires current.Valid()
    requires child.Valid()
    requires !(child.keys[0] in current.Contents)
    requires !current.isLeaf
    ensures fresh(newNode)
    ensures !current.isLeaf && !newNode.isLeaf
    ensures current.Valid() && newNode.Valid()
    ensures child.Valid()
    ensures old(child.Contents) == child.Contents // same
    ensures current.HalfFull() && newNode.HalfFull()
    ensures current.keyNum > 0 && newNode.keyNum > 0
    ensures current.keys[current.keyNum-1] < newNode.keys[0]
    ensures old(current.Contents) < (current.Contents + newNode.Contents) // all values kept
    ensures current.Contents !! newNode.Contents // disjoint
    ensures old(current.Contents)+child.Contents == current.Contents + newNode.Contents // all values kept
```

Figure 15: SplitNode specifications

In the SplitNode method, *current* represents the node being split, *child* is the node causing the split, as its first key should be inserted (along with the pointer towards this node), and it returns the newly created *newNode*. The method has specifications similar to the SplitLeaf method. It modifies only the *current* node, its *Repr* (meaning adding nodes to its subtree), and the *newNode*, which does not have to be explicitly specified, as this node is created in this method. The method requires that the parent and child nodes are both valid, and that the parent does not already contain the value being inserted (the first key of the child). Since we are dealing with internal nodes, the parent node cannot be a leaf. After the execution, both modified nodes, *current* and *newNode*, should be valid, and the *child* node should remain valid. The *Contents* variable of the *child* node should not change, and as in SplitLeaf, the nodes being split should

be half full, with their *keyNum* variables greater than zero, and the last key in *current* should be less than the first key in *newNode*. All values that were in the *Contents* variable of the node being split should be retained, meaning that the *Contents* set before the execution should be a subset of the union of *Contents* of nodes *current* and *newNode*. These *Contents* sets should be disjoint, and if we add the *Contents* of the child node to the *Contents* of the node being split, we should have the same values as the union of *Contents* of *current* and *newNode* after the execution.

We can see that this algorithm follows the structure of a B+ tree, handling leaf and internal node insertions and splits as needed. The recursion ensures that the insertion process cascades through the tree, and the splitting of nodes maintains the balanced and sorted properties of the B+ tree.

4 Discussion and Future Work

The main problem in the verification process for our B+ tree implementation is that it involves considerable time. This required adjustments to Visual Studio Code's time limit setting. Extending the limit to 100 seconds has proven effective for some functions but not universally. A practical approach to somewhat mitigate this problem is to comment out methods that you are not currently working on. A similar challenge was noted in [9], where the authors were verifying Red-Black trees in Dafny and observed a 50-second verification time for their entire file, which is shorter than for some of our functions. We believe that the complexity of verifying B+ trees comes from the use of arrays in every node. Most methods involve array manipulation, necessitating the incorporation of for loops with invariants, asserts, and predicates for node verification. The added complexity comes from the fact that these arrays are often partially filled. Consequently, iterating from 0 to the number of keys entails verifying that this variable remains within the length of the array, defined by the constant *ORDER*.

We're still getting the hang of Dafny, and we've considered leveraging existing functionalities like triggers to simplify the verification process. Understanding why Dafny struggles with certain proofs has been a challenge for us. While the idea of an IDE in Visual Studio Code with a Dafny plugin providing real-time feedback is appealing, in reality, the feedback isn't always as immediate or informative as we'd hoped. Most error messages simply indicate that Dafny couldn't conclusively validate an assert, leaving room for interpretation rather than definitively declaring it is false.

Initially, we placed significant reliance on Dafny, assuming it could handle more complex reasoning tasks than it actually could. As our verification process progressed, we realized that guiding the verification process is crucial, but we observed that excessive reliance on assert statements might also extend the verification time. A more effective strategy involves breaking down the code into smaller parts, necessitating the creation of additional specifications. This approach not only accelerates the verification process but, in some cases, makes

it feasible.

Another valuable insight we gained is the utility of lemmas in reducing verification time. An example of a lemma is presented in Figure 11, and here, Figure 16 introduces another lemma consisting of a single assert. Leveraging such lemmas in the code, instead of asserts, expedited the verification of the InsertIntoSorted method.

```
lemma DistributiveGreater(a: seq<int>, b:seq<int>, k:int, key:int)
    requires |a| + 1 == |b|
    requires 0 <= k <= |a|
    requires b == a[..k] + [key] + a[k..]
    requires forall j :: 0 <= j < |a| ==> a[j] > 0
    requires key > 0
    ensures forall i :: 0 <= i < |b| ==> b[i] > 0
{
    assert forall j :: 0 <= j < |b| ==> b[j] > 0;
}
```

Figure 16: DistributiveGreater lemma

It's important to note that our observations about verification time might not be entirely accurate. Another challenge with Dafny and VS Code is occasional instances where Dafny stops working and requires a restart. Restarting VS Code or even the whole system sometimes influences the verification speed, even without making any code changes. Unfortunately, VS Code does not provide output for verification time, so our conclusions are based on potential timeouts when setting the Timeout Limit to a specific duration.

Figure 17 provides examples illustrating Dafny's ability to reason about sets, specifically focusing on set cardinalities and the equality in the number of elements between a set and a sequence of distinct numbers. This scenario mirrors our experience with the *keys* array, represented as a sequence generated by $keys[..keyNum]$, and the *Contents* set. When manually creating both a set and a sequence with the same elements, Dafny correctly recognizes that they share the same number of elements, as demonstrated in the first part of the figure.

However, when constructing a set as a collection of elements from the sequence, even with additional assertions confirming the equivalence of set and sequence elements and ensuring that the sequence elements are all distinct, Dafny struggles to deduce the equal number of elements or that the set contains precisely three elements. Another aspect of the example illustrates that if another set is created in a same manner, Dafny successfully acknowledges the equal number of elements between these two sets.

Conversely, if the other set is manually generated with identical elements, Dafny encounters difficulty verifying the equality in the number of elements. Here, the set_membership_implies_cardinality lemma (Figure 18) proves beneficial. This lemma accepts two sets as input, requires that all elements from one set are present in the other, and guarantees they share the same number

```
// if you manualy create set and sequence with same elements, |s|==|t| works
var t: seq<int> := [1, 2, 3];
var s: set<int> := {1, 2, 3};
assert |s| == 3;
assert |s| == |t|;

// but if you create set from the sequence with distinct elements it does not understand that |s|==|t|
// Dafny has problems when reasoning about set sizes ==>
s := set x | x in t;
assert forall x :: x in t ==> x in s;
assert forall x :: x in s ==> x in t;
assert forall x :: x in s <==> x in t;
assert forall i, j :: 0 <= i < |t| && 0 <= j < |t| && i !=j ==> t[i] != t[j];
assert |t| == 3;
// assert |s| == |t|; // not verifying
// assert |s| == 3; // not verifying

// other expriments
set_memebrship_implies_cardinality(s, set x | x in t);  // s and the other argument is the same thing
var s2 : set<int> := set x | x in t;
assert |s| == |s2|;

s2 := {1, 2, 3};
// assert |s| == |s2|; // may not hold
set_memebrship_implies_cardinality(s, s2);
assert |s| == |s2|; // after lemma it holds
```

Figure 17: Dafny verification with sets example

of elements. The lemma relies on a recursive helper lemma, which, in addition to the two sets, takes the size of one set as input, requiring it is greater than or equal to zero and equal to the number of elements in one of the sets. The recursive lemma employs a decrease clause based on the size of the set. It takes one element from a set, and calls the recursion with the sets from which this element is removed, and the size of the set is decreased by one.

While similar lemmas may be conceivable to demonstrate equality between a set and a sequence, our attempts to formulate one were not successful. Throughout our research, we noted that others encounter challenges with sets in Dafny, as evident in various Stack Overflow threads, such as this one and this one.

In Visual Studio Code, we observed the presence of an experimental feature known as Dafny Caching Policy. This feature is designed to enable Dafny to reuse prior verification results while a source file is being edited. The available options include No caching (the default setting), Basic caching, Advanced caching, and Rewrites the whole file after each change. Due to our late discovery of this experimental feature and our prior experiences with Dafny restarts and unexpected crashes, we did not explore this option. Nevertheless, we acknowledge its existence and speculate that it might address some of the timeout issues we encountered, particularly if it transitions from an experimental status to a stable and reliable feature in the future.

Next step in our research would be to finish the verification of insert method and its helper methods. After this, the implementation of delete and query range methods, which is essential to achieve a complete B+ tree node implementation. Although our initial plan included writing the delete function, we encountered

```
lemma set_memebrship_implies_cardinality_helper<A>(s: set<A>, t: set<A>, s_size: int)
  requires s_size >= 0 && s_size == |s|
  requires forall x :: x in s <==> x in t
  ensures |s| == |t|
  decreases s_size {
  if s_size == 0 {
  } else {
    var s_hd;
    // assign s_hd to a value *such that* s_hd is in s (see such_that expressions)
    s_hd :| s_hd in s;
    set_memebrship_implies_cardinality_helper(s - {s_hd}, t - {s_hd}, s_size - 1);
  }
}


lemma set_memebrship_implies_cardinality<A>(s: set<A>, t: set<A>)
  requires forall x :: x in s <==> x in t
  ensures |s| == |t| {
  set_memebrship_implies_cardinality_helper(s, t, |s|);
}
```

Figure 18: Lemmas about the set cardinality equality

challenges with the insert and find methods, preventing us from progressing to the delete function.

The delete method introduces additional complexities compared to the insert method, as it involves handling various edge cases. Similar to insert, it requires maintaining the correct number of keys in each node. However, instead of splitting a node, the delete method involves merging a node with one of its siblings. This process can propagate changes upward in the tree, potentially altering the parent node and even causing a reduction in tree height if the root is replaced with its single remaining child.

In addition to managing structural changes, the delete function also necessitates addressing storage deallocation. Unfortunately, we haven't explored how Dafny handles this aspect. For guidance in implementing the delete method, we recommend referring to [3], a paper that initially introduced pseudocode, algorithm and flowchart for deletion in B+ trees.

Regarding the query range method, let us first define it. This method allows for retrieving all values within a specified range. It is typically defined by a lower and upper bound, and the query retrieves all values falling within that range. However, its apparent simplicity can be deceiving. A verified version of this method was first presented in 2022 [11]. The complexity arises from the use of two entirely independent predicates that describe overlapping memory, creating a challenge in separation logic. These predicates entail the explicit formulation of the linked list view of the leaves and the requirement to ensure that iterating through the leaves does not alter the entire tree.

Abstract definitions, which don't specify concrete memory locations, are generally beneficial for reducing specification and proof obligations. However,

in this case, it becomes necessary to establish that the memory locations are the same in both predicates. In the work of [11], the authors addressed this challenge by dividing the tree into a trunk and a fringe, defining the fringe as a concatenation of all fringes in its subtrees.

The query range method holds significant importance due to the B+ tree's inherent property of allowing sequential access to data, making it a highly desirable feature for database systems [3]. Properly designed and implemented, this method contributes to the overall performance and versatility of B+ trees in managing and retrieving data.

In conclusion, having a verified implementation of a B+ tree, along with all its methods, is crucial for several reasons:

- A verified implementation ensures that the B+ tree behaves as expected, preserving the integrity of the data structure. This is particularly important in scenarios where the B+ tree is a fundamental component of a database management system.
- B+ trees are widely used in database systems for efficient data organization. A verified implementation instills confidence in the reliability of the data structure, reducing the risk of unexpected behavior or errors in critical systems.
- A verified B+ tree implementation serves as a solid starting point for future development and enhancements. Developers can build upon the verified foundation with confidence, knowing that the existing functionality meets specified correctness criteria.

Having this implementation done in Dafny is beneficial for several reasons:

- Dafny's formal methods allow for the clear and precise specification of preconditions, postconditions, and invariants. This makes it easier to understand the verification process and ensures that the specified properties are upheld throughout the implementation.
- Dafny's verified implementation can be easily exported to other languages or integrated into existing systems. The assurance of correctness provides a solid foundation for interoperability, enabling seamless interaction with components written in different languages.

References

[1] Bayer, R., & McCreight, E. (1970, November). Organization and maintenance of large ordered indices. In Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control (pp. 107-141).

[2] Fielding, E. (1980). The specification of abstract mappings and their implementation as B Trees. Oxford University Computing Laboratory, Programming Research Group.

[3] Jannink, J. (1995). Implementing deletion in B+-trees. ACM Sigmod Record, 24(1), 33-38.

[4] Comer, D. (1979). Ubiquitous B-tree. ACM Computing Surveys (CSUR), 11(2), 121-137.

[5] Malecha, G., Morrisett, G., Shinnar, A., & Wisnesky, R. (2010, January). Toward a verified relational database management system. In Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (pp. 237-248).

[6] Sexgon, A. & Thielecke, H. (2008). Reasoning about B+ trees with operational semantics and separation logic. Electronic Notes in Theoretical Computer Science, 218:355–369.

[7] Ernst, G., Schellhorn, G., & Reif, W. (2011). Verification of B+ trees: an experiment combining shape analysis and interactive theorem proving. In Software Engineering and Formal Methods: 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings 9 (pp. 188-203). Springer Berlin Heidelberg.

[8] Leino, K. R. M. (2010, April). Dafny: An automatic program verifier for functional correctness. In International conference on logic for programming artificial intelligence and reasoning (pp. 348-370). Berlin, Heidelberg: Springer Berlin Heidelberg.

[9] Peña, R. (2020). An assertional proof of red–black trees using Dafny. Journal of Automated Reasoning, 64(4), 767-791.

[10] Mündler, N. (2021, February). A Verified Imperative Implementation of B-Trees. Archive of Formal Proofs.

[11] Mündler, N., & Nipkow, T. (2022, September). A Verified Implementation of B+-Trees in Isabelle/HOL. In International Colloquium on Theoretical Aspects of Computing (pp. 324-341). Cham: Springer International Publishing.

[12] Galles, G. (n.d.). B+ Tree Visualization. University of San Francisco. `https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html`