Towards Data Structure Synthesis

A thesis presented
by
Andrew Lim

To
The Department of Computer Science

Harvard University
Cambridge, Massachusetts
May 2021
Towards Data Structure Synthesis

Abstract

Designing efficient data structures and implementing them correctly is a difficult and costly process. The Data Calculator is a design engine that partially solves this problem by automatically synthesizing the most efficient data structure design for a target workload and hardware. In this thesis, we build on the Data Calculator by demonstrating how to programmatically generate formal data structure specifications of Data Calculator designs. These specifications are a key stepping stone towards the ultimate goal of data structure synthesis — the automated construction of data structure implementations that provably satisfy data structure specifications. Correct and efficient data structure implementations can thus be generated by a data structure synthesizer that uses specifications of efficient data structure designs produced by the Data Calculator.
## Contents

1 Introduction 1

2 Overview 5
   2.1 Data Calculator Data Structure Designs 6
   2.2 Data Calculator Design Primitives 7
   2.3 Node Definitions and Functional Specifications 10
   2.4 Per-Element Specification Generation 12
   2.5 Separation Logic and Notation 13

3 Terminal Nodes 16
   3.1 Terminal Element Parameters 17
   3.2 Terminal Node Invariant Specification 19
   3.3 Data Size and Capacity Metadata (9) 20
   3.4 Utilization and Navigation (5, 22, 23) 21
   3.5 Key-Value Layout and Ordering (1, 2, 3, 4, 10) 22

4 Non-Terminal Nodes 28
   4.1 Non-Terminal Node Parameters 29
   4.2 Non-Terminal Node Invariant Specification 31
   4.3 Sub-Block Metadata (9, 11, 19) 33
   4.4 Navigation (22, 23) 35
   4.5 Zone Map Filters (7, 8, 9, 10, 19) 36
   4.6 Sub-Block Layout and Link Correctness (4, 9, 10, 12, 15, 16, 19, 20) 39
   4.7 Sub-Block Capacity (11) 46
   4.8 Sorted Invariant (10) 46

5 Data Structure Operations 47
   5.1 Get 48
   5.2 Delete 49
   5.3 Insert 51

6 Conclusion 54

References 56
To Alice
Who told me not to do this
But I didn’t listen

And here we are.
Acknowledgments

First and foremost, I am immensely grateful to my thesis advisor, Professor Stephen Chong, for the invaluable guidance and sincere encouragement that he so generously provides.

To Professor Stratos Idreos, thank you so much for agreeing to read my thesis and for your thoughtful advice. Taking your classes inspired my interest in data systems.

To Eric and Crystal, I really appreciate all the time you spent chatting with me and sharing your insights.

To Subarna and Wilson from the DASLab, thank you for clarifying my understanding of the Data Calculator.
Data structures are a fundamental software component for data-intensive applications which are becoming increasingly commonplace with the proliferation of big data. However, developing highly efficient data structures is incredibly complex and costly, in part because the performance implications of different design choices are difficult to determine.
by humans. Consequently, candidate data structure designs need to be implemented and tested on the necessary workloads and hardware, in order to accurately gauge their viability. This lengthy iterative process is compounded by the significant engineering time and effort required to implement a data structure design and to ensure that it is error-free.

Our overarching vision is the automatic generation of efficient and correct data structure implementations. This goal has been partially realized by the Data Calculator — a design engine that can automatically synthesize data structure designs from a set of design primitives and estimate their performance. In short, the Data Calculator is able to determine the most efficient data structure design for a particular use case. Therefore, the remaining hurdle and natural next step is to generate correct data structure implementations from Data Calculator designs.

In this thesis, we demonstrate how data structure specifications can be programmatically created from Data Calculator designs. These high-level specifications formally describe the correct data structure behaviour and are a key stepping stone towards data structure synthesis — the automated construction of a data structure implementation that provably satisfies a given data structure specification.

Data structure synthesis is an apt approach for this problem as data structure specifications are significantly easier to directly generate than the implementations which have many possibilities. Take for instance a B-Tree — a self-balancing tree data structure that is
widely used in databases. Generating the rebalancing code is not straightforward as there are many different valid B-Tree rebalancing strategies. Furthermore, since the Data Calculator design primitives are primarily intended for estimating performance, they may omit implementation details that are irrelevant to costing data structure designs. For instance, the notion of balance in a B-Tree is expressed by the Data Calculator parameters, but the rebalancing process is left entirely uncharacterized.

In contrast, simply specifying that a B-Tree must always be balanced is sufficient to synthesize a B-Tree implementation with valid rebalancing code. This is because implementations that do not perform rebalancing correctly will violate the balanced invariant and thus will be rejected by the data structure synthesizer. In the partial B-Tree specification below, we show that this balanced invariant can be easily specified with the single underlined constraint that enforces the equality of child node data capacities:

\[
\text{children}_{B_{\text{tree}}}(t, i, n) \triangleq (i > n - 1) \lor \\
\quad (i < n - 1 \land t->child[i]->cap = t->child[i+1]->cap \\
\quad \land \text{children}_{B_{\text{tree}}}(t, i + 1, n))
\]

In Chapter 2, we give an overview of the data structure specification generation process and introduce separation logic which allows us to reason about the correctness of programs. We also provide further background about the Data Calculator and its data structure designs. In Chapters 3, 4, and 5, we show how to generate data structure definitions, invari-
ant specifications, as well as functional specifications for the 3 main data structure operations: Get, Delete, and Insert. Finally, in Chapter 6, we discuss how related work on data structure synthesis compares to our approach as well as future directions to explore.
In this chapter, we walk through the process of generating the data structure specification for a Data Calculator data structure design that describes a singly linked list. Through this example, we explain the relevant Data Calculator concepts, showcase the different specification components, and introduce separation logic as well as the notation that we use.
2.1 Data Calculator Data Structure Designs

A Data Calculator data structure design is a hierarchy of one or more elements. Each element describes a particular node in the data structure according to the Data Calculator design primitives which we discuss in the following section. A singly linked list has the following Data Calculator design:

**Linked List Partition (LLP) → Data Page**

This design indicates that a singly linked list is *logically* represented by a single **LLP** element which has logical sub-blocks or child nodes that are **Data Page** elements. This two-element characterization of a singly linked list is necessary in order to specify distinct *physical* aspects of the data structure. The **LLP** element describes the partitioning of the data into an unbounded number of child nodes of fixed size that can only be accessed by following inter-node pointers starting from the first sub-block. On the other hand, the **Data Page** element details the layout and ordering of the data contained within a data node, which may include more than one key-value pair. Consequently, despite the different logical conceptualization, the Data Calculator singly linked list is physically congruent with an archetypal singly linked list. As depicted in Figure 2.1, the actual nodes are scattered in memory and the LLP node is effectively a pointer to the first node in a singly linked chain of data page nodes.
Figure 2.1: Physical Representation of a Singly Linked List Data Calculator Design

Note that by definition a Data Calculator data structure design must have at least one terminal element, but may have zero or more non-terminal elements. In the case of the singly linked list, **Data Page** is a terminal element, while **LLP** is the root non-terminal element. For this thesis, we assume that all sub-blocks of an element are of the same type and hence every design is a linear hierarchy of elements which ends with a single terminal element. It is also possible for a non-terminal element to be recursively connected to itself, but we do not consider such designs in this thesis.

2.2 **Data Calculator Design Primitives**

The full physical description of a Data Calculator node is given by the settings of 21 design primitives which correspond to parameters 1-21 in Table 2.1. Each design primitive represents a fundamental design choice and each setting of a design primitive denotes a different tuning of that aspect of the data structure design.⁵
<table>
<thead>
<tr>
<th>Parameter</th>
<th>LLP</th>
<th>Data Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Retention</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Value Retention</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>Key-Value Layout</td>
<td>N.A.</td>
<td>COLUMNAR</td>
</tr>
<tr>
<td>Intra-Node Access</td>
<td>HEAD</td>
<td>DIRECT</td>
</tr>
<tr>
<td>Utilization</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>Bloom Filters</td>
<td>OFF</td>
<td>OFF</td>
</tr>
<tr>
<td>Zone Map Filters</td>
<td>OFF</td>
<td>OFF</td>
</tr>
<tr>
<td>Filters Memory Layout</td>
<td>N.A.</td>
<td>N.A.</td>
</tr>
<tr>
<td>Fanout / Radix</td>
<td>UNLIMITED</td>
<td>TERMINAL(256)</td>
</tr>
<tr>
<td>Key-Fence Partitioning</td>
<td>BACKWARD</td>
<td>FORWARD</td>
</tr>
<tr>
<td>Sub-Block Capacity</td>
<td>FIXED(256)</td>
<td>N.A.</td>
</tr>
<tr>
<td>Immediate Node Links</td>
<td>NEXT</td>
<td>NONE</td>
</tr>
<tr>
<td>Skip Node Links</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>Area Links</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>Sub-Block Physical Location</td>
<td>POINTED</td>
<td>N.A.</td>
</tr>
<tr>
<td>Sub-Block Physical Layout</td>
<td>SCATTER</td>
<td>N.A.</td>
</tr>
<tr>
<td>Sub-Blocks Homogeneous</td>
<td>TRUE</td>
<td>N.A.</td>
</tr>
<tr>
<td>Sub-Block Consolidation</td>
<td>FALSE</td>
<td>N.A.</td>
</tr>
<tr>
<td>Sub-Block Instantiation</td>
<td>LAZY</td>
<td>N.A.</td>
</tr>
<tr>
<td>Sub-Block Links Layout</td>
<td>SCATTER</td>
<td>N.A.</td>
</tr>
<tr>
<td>Recursion</td>
<td>NO</td>
<td>N.A.</td>
</tr>
<tr>
<td>Sibling Node Links</td>
<td>NO</td>
<td>NEXT</td>
</tr>
<tr>
<td>Node Zone Map Filter</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

Table 2.1: Parameter Settings for the Nodes in a Singly Linked List Data Calculator Design.
For example, 9. **Fanout / Radix** defines the node’s fanout or maximum number of sub-blocks. We see that **LLP** has unlimited fanout which means that there can be arbitrarily many **Data Page** nodes. In contrast, the setting for **Data Page** is **terminal(256)** which means that it is a terminal element that does not have any sub-blocks and simply stores data up to 256 records.

Since some parent node primitives directly influence their sub-blocks, we reproduce the relevant primitive settings in the child node for convenience via parameters 22 and 23. For instance, 22. **Sibling Node Links** takes on the same setting as the parent node’s 12. **Immediate Node Links** which indicates whether its sub-blocks have pointers between them as well as the direction of the pointers if they exist. In the table, we see that the setting of 12. **Immediate Node Links** for **LLP** is the same as 22. **Sibling Node Links** for **Data Page**.

For this thesis, we do not consider all possible settings of the parameters when generating data structure specifications. For example, 6. **Bloom Filters** and 13. **Skip Node Links** are set to no by default as specifying probabilistic programs is out of the scope of this thesis which focuses on heap-manipulating programs. In subsequent chapters, we make sure to enumerate the chosen domains of parameter settings and to explain the meaning of the different parameters as we lay out the specification generation rules. For a full description and explanation of the Data Calculator design primitives, we direct the reader to the Data Calculator Technical Report. 6
2.3 Node Definitions and Functional Specifications

The data structure specification that is generated from a Data Calculator design has two main components. The first component is the node definitions of the elements in the design. Each node definition is generated based on the corresponding element’s parameter settings. As valid C++ class definitions, they form part of the data structure implementation, in addition to representing the state that is stored in each node. Class members, which we term definition fields, are referenced in other parts of the data structure specification to reason about the values at those memory locations. Below, we display the node definitions generated from the parameter settings of the singly linked list elements shown in Table 2.1. Note that we assume all keys and values stored in the data structure have type int.

```cpp
class LinkedListPartitionNode {
  unsigned int sbinit; // current number of initialized sub-blocks
  unsigned int fan; // fanout or the maximum number of sub-blocks
  unsigned int cap; // total data capacity of all child nodes
  DataPageNode* sbhd; // pointer to first sub-block
};

class DataPageNode {
  unsigned int sz; // current number of key-value pairs
  static const unsigned int cap = 256; // max number of key-value pairs
  int kvps[cap * 2]; // array containing key-value pairs
  DataPageNode* nxt; // pointer to sibling node
};
```
The second component of the data structure specification is the functional specifications for the data structure operations Get, Insert, and Delete. These functional specifications are Hoare triples which have the form $\{P\} F \{Q\}$, where $F$ is a data structure operation function, while $P$ and $Q$ are logical assertions. A Hoare triple denotes that if the pre-condition $P$ holds true before the execution of the function $F$, then the post-condition $Q$ must hold true after $F$ has finished. The Delete Hoare triple for the singly linked list example is:

$$\{\text{LLInvariant}(t, K_1, V_1)\}$$

$$\text{void del(int key)}$$

$$\{\text{deleted(key, } K_1, K_2, V_1, V_2) \land \text{LLInvariant}(t, K_2, V_2)\}$$

The LLInvariant predicate is the data structure invariant which specifies that there is a valid singly linked list at the memory location $t$. As such, it must hold true in both the pre-condition and the post-condition. The variables $K$ and $V$ are nested, ordered lists that respectively contain all the keys and values in the data structure. The nesting of the lists corresponds to the logical node hierarchy and thus describes the distribution of key-value pairs across nodes. Observe that there are distinct $K$ and $V$ lists in the pre-condition and post-condition since the operation modifies the data in the data structure and possibly the organization of the nodes. The deleted predicate thus specifies list manipulations that reflect the correct change in the logical data structure state as a result of the Delete operation. Therefore, to construct a Hoare Triple, we need to generate the data structure invariant specification as well as an appropriate list manipulation predicate.
2.4 Per-Element Specification Generation

Node invariant specifications and list manipulation predicates are independently generated for each element in the design based on their respective parameter settings. These individual specifications and predicates are then recursively combined to form the data structure invariant specification and overall list manipulation predicate for the Hoare Triple. For example, we see below that the data page node invariant specification \( \text{nodeInv}_{DP} \) is referenced in the LLP node invariant specification \( \text{nodeInv}_{LLP} \) to assert the correctness of the latter's child nodes. As the root element of the design, \( \text{nodeInv}_{LLP} \) is thus equivalent to the aforementioned \( \text{LLInvariant} \).

\[
\text{nodeInv}_{DP}(t, K, V) \triangleq t->sz \leq t->cap \land ((\text{list}(\&(t->keys), t->cap - t->sz, t->cap, K) \land \\
\text{isNull}(\& (t->keys), 0, t->cap - t->sz)) \ast \\
(\text{list}(\& (t->vals), t->cap - t->sz, t->cap, V) \land \\
\text{isNull}(\& (t->vals), 0, t->cap - t->sz)))
\]

\[
\text{nodeInv}_{LLP}(t, K, V) \triangleq t->cap = C \land t->sbinit \leq t->fan \land \\
\text{subBlocks}_{LLP}(t, t->sbhd, 0, t->sbinit, K, V, C)
\]

\[
\text{subBlocks}_{LLP}(t, c, i, n, K, V, C) \triangleq (i = n \land K = \{ [] \} \land V = \{ [] \} \land C = 0 \land \text{emp}) \lor \\
(i < n \land K = \{ [K_1] \} \parallel K_2 \land V = \{ [V_1] \} \parallel V_2 \land \\
C = c->cap + C_2 \land c->cap = 256 \land \\
((i = n - 1 \land c->nxt = \text{null}) \lor (i < n - 1)) \land \\
(\text{nodeInv}_{DP}(c, K_1, V_1) \ast \text{subBlocks}_{LLP}(t, t->nxt, i + 1, n, K_2, V_2, C_2)))
\]
2.5 Separation Logic and Notation

Node invariant specifications are generated are separation logic formulas. The heart of separation logic is the separating conjunction $P \ast Q$ which asserts that $P$ and $Q$ hold true in separate parts of memory. To illustrate the use of this operator, let us take a closer look at the definition of subBlocks_{LLP} above which applies the separating conjunction on nodeInv_{DP} and recursively on itself:

$$\text{subBlocks}_{LLP}(t, c, i, n, K, V, C) \triangleq \ldots (\text{nodeInv}_{DP}(c, K_1, V_1) \ast \\
\quad \text{subBlocks}_{LLP}(t, t\rightarrow\text{nxt}, i + 1, n, K_2, V_2, C_2)))$$

Essentially, subBlocks_{LLP} uses the separating conjunction to recursively assert that each of its child nodes (along with any nodes underneath them) must exist in in separate portions of memory. This makes sense as the different sub-trees of the data structure node hierarchy cannot be allocated in overlapping segments of memory.

Similarly, the list predicate used in nodeInv_{DP} above recursively specifies an allocated contiguous memory segment starting from memory location $a+i$ to $a+n-1$. The notation $[a, n] \mapsto [a_0, \ldots, a_{n-1}]$ denotes an allocated contiguous memory segment of length $n$ which has value $a_i$ at memory location $a+i$.

$$\text{list}(a, i, n, L) \triangleq (i = n \land L = \{\} \land \text{emp}) \lor \\
\quad (i < n \land L = \{x\} \parallel L_1 \land ([a + i, 1] \mapsto [x] \ast \text{list}(a, i + 1, n, L_1)))$$
We see that the separating conjunction enforces that the allocated memory locations in the memory segment do not overlap. In the base case of the recursion \((i = n)\), emp denotes a memory segment which has nothing allocated. This helps to specify that there should not be more memory allocated than what is necessary for the data structure.\(^8\)

Another important piece of notation in the list predicate is ordered list notation which we employ as a logical representation of the physical state. \([a_0, a_1, \ldots, a_n]\) denotes an ordered list of values and \(\parallel\) is a binary operator which concatenates two lists. Hence, list additionally specifies that the values in the allocated memory segment correspond to the values in \(L\).

We summarize other notation that we use in this thesis in Table 2.2 below for the reader’s reference. The notation for describing memory locations is the same as C++ syntax and the list indexing notation uses Python slice syntax.

Looking ahead, in Chapters 3 and 4, we outline node definition and node invariant specification generation procedures for terminal and non-terminal elements respectively. In Chapter 5, we showcase the appropriate list manipulation predicates for each data structure operation to complete the Hoare Triple functional specifications.
<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨a₀, a₁⟩</td>
<td>Denotes a tuple of values.</td>
</tr>
<tr>
<td>{a₀, a₁, ..., aₙ}</td>
<td>Denotes a multiset.</td>
</tr>
<tr>
<td>{a₀, a₁, ..., aₙ}</td>
<td>Denotes an ordered list of values.</td>
</tr>
<tr>
<td>{a₀, a₁, ..., aₙ}</td>
<td>Denotes a multiset containing values a₀, a₁, ..., aₙ.</td>
</tr>
<tr>
<td>flatten({...})</td>
<td>For L = {...}, flatten(L) is a flat version of L.</td>
</tr>
<tr>
<td>min({...})</td>
<td>For L = {...}, min(L) is the minimum value in flatten(L).</td>
</tr>
<tr>
<td>max({...})</td>
<td>For L = {...}, max(L) is the maximum value in flatten(L).</td>
</tr>
<tr>
<td>{...}</td>
<td></td>
</tr>
<tr>
<td>(i)</td>
<td>For L = {{a₀, a₁, ..., aₙ}}, L(i) denotes aᵢ for 0 ≤ i ≤ n.</td>
</tr>
<tr>
<td>(i : i)</td>
<td>For L = {{a₀, a₁, ..., aₙ}}, L(i : i) denotes {{a₀, ..., aᵢ₋₁}} for 0 &lt; i ≤ n.</td>
</tr>
<tr>
<td>(i : i)</td>
<td>For L = {{a₀, a₁, ..., aₙ}}, L(i : i) denotes {{aᵢ, ..., aₙ}} for 0 ≤ i ≤ n.</td>
</tr>
<tr>
<td>P * Q</td>
<td>Separating conjunction; P and Q hold true in separate parts of memory.</td>
</tr>
<tr>
<td>[a, n] ↦ [a₀, ..., aₙ₋₁]</td>
<td>Denotes an allocated memory segment of length n which has value aᵢ at memory location a+i.</td>
</tr>
<tr>
<td>emp</td>
<td>Denotes a memory segment which has nothing allocated.</td>
</tr>
<tr>
<td>null</td>
<td>Denotes a null value.</td>
</tr>
<tr>
<td>t→var</td>
<td>Denotes the value of the definition field var.</td>
</tr>
<tr>
<td>&amp;(t→var)</td>
<td>Denotes the memory location of t→var.</td>
</tr>
<tr>
<td>t→arr[i]</td>
<td>Denotes the value at the memory location &amp;(t→var)+i.</td>
</tr>
</tbody>
</table>

Table 2.2: Notation Cheatsheet.
Terminal Nodes

Terminal nodes refer to nodes that are described by the terminal element of a Data Calculator design. As logical leaves of the data structure, the primary purpose of terminal nodes is to store data in a variety of physical layouts and orderings. For instance, keys and values may be stored separately in two segments of memory rather than in a single contigu-
ous chunk, or the key-value pairs may be kept in sorted order. Terminal nodes may also keep additional metadata and pointers to aid in navigation across nodes. In this chapter, we demonstrate how terminal node definitions and invariant specifications can be procedurally generated from the parameters of a terminal element.

3.1 Terminal Element Parameters

Recall that a Data Calculator element describes a node according to the settings of 21 different design primitives. These design primitives correspond to parameters 1-21 in Table 3.1. Parameters 22 and 23 are added for convenience to reproduce parent design primitive settings that influence the element. For each parameter in the table, we list the settings that we consider for the generation of terminal node definitions and invariant specifications. The parameters that directly affect the generated output are highlighted in grey.

Note that by definition there is only one possible setting for 9. Fanout / Radix for terminal elements which is TERMINAL(CAP). Additionally, some settings, such as ROW-GROUPS(GSIZE), are parameterized by an additional value which we represent as a meta-variable. Monospace font indicates that the meta-variable is used as a value in the node definition or to represent a physical value in memory, while sans serif font indicates that the meta-variable is used as a logical value.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Key Retention</td>
<td>YES</td>
</tr>
<tr>
<td>2  Value Retention</td>
<td>YES</td>
</tr>
<tr>
<td>3  Key-Value Layout</td>
<td>COLUMNAR, ROW-WISE, ROW-GROUPS(GSIZE)</td>
</tr>
<tr>
<td>4  Intra-Node Access</td>
<td>DIRECT</td>
</tr>
<tr>
<td>5  Utilization</td>
<td>$\geq 50%$, NONE</td>
</tr>
<tr>
<td>6  Bloom Filters</td>
<td>OFF</td>
</tr>
<tr>
<td>7  Zone Map Filters</td>
<td>OFF</td>
</tr>
<tr>
<td>8  Filters Memory Layout</td>
<td>N.A.</td>
</tr>
<tr>
<td>9  Fanout / Radix</td>
<td>TERMINAL(CAP)</td>
</tr>
<tr>
<td>10 Key-Fence Partitioning</td>
<td>FORWARD, BACKWARD, SORTED</td>
</tr>
<tr>
<td>11 Sub-Block Capacity</td>
<td>N.A.</td>
</tr>
<tr>
<td>12 Immediate Node Links</td>
<td>NONE</td>
</tr>
<tr>
<td>13 Skip Node Links</td>
<td>NONE</td>
</tr>
<tr>
<td>14 Area Links</td>
<td>NONE</td>
</tr>
<tr>
<td>15 Sub-Block Physical Location</td>
<td>N.A.</td>
</tr>
<tr>
<td>16 Sub-Block Physical Layout</td>
<td>N.A.</td>
</tr>
<tr>
<td>17 Sub-Blocks Homogeneous</td>
<td>N.A.</td>
</tr>
<tr>
<td>18 Sub-Block Consolidation</td>
<td>N.A.</td>
</tr>
<tr>
<td>19 Sub-Block Instantiation</td>
<td>N.A.</td>
</tr>
<tr>
<td>20 Sub-Block Links Layout</td>
<td>N.A.</td>
</tr>
<tr>
<td>21 Recursion</td>
<td>NO</td>
</tr>
<tr>
<td>22 Sibling Node Links</td>
<td>NEXT, PREV, BOTH, NO</td>
</tr>
<tr>
<td>23 Node Zone Map Filter</td>
<td>MIN, MAX, BOTH, NO</td>
</tr>
</tbody>
</table>

Table 3.1: Relevant Parameters and Settings for Generating Terminal Node Definitions and Invariant Specifications.
3.2 Terminal Node Invariant Specification

We define the terminal node invariant specification as follows:

\[ \text{nodeInv}_{\text{t}}(t, K, V) \triangleq \bigwedge_{c \in \text{termInv}} c \]

\text{termInv} is a set of logical constraints that describe properties of the terminal node’s state that hold true regardless of any prior sequence of data structure operations. In other words, the physical state of a valid terminal node must always satisfy all of these constraints. \( t \) is the start memory location of the terminal node, while \( K \) and \( V \) are logical ordered lists of all the keys and values in the terminal node respectively. Note the recurring font convention which uses monospace font for variables related to the physical node state and italicized font for logical variables. The simple structure of this specification reflects the fact that terminal nodes are relatively uncomplicated as they do not have any sub-blocks and hence use less parameters and settings than non-terminal nodes. In Chapter 4, we shall examine more complex node invariant specifications for non-terminal nodes.

To showcase the generation of node definition fields and invariant constraints, we discuss separate subsets of parameters that affect similar aspects of the node state. For parameters that interact with each other, we make sure to indicate the generated output for different setting combinations.
3.3 Data Size and Capacity Metadata (9)

First, let us consider the definition fields and node invariant constraints that all terminal nodes should have by default. As mentioned previously, the parameter 9, Fanout / Radix is set to terminal(CAP) by default. The meta-variable CAP specifies the maximum number of key-value pairs that can be stored in the terminal node. This implies that a terminal node must keep track of the amount of data stored to check against the node data capacity in order to prevent overflow. Hence, the default definition fields are:

```cpp
class TerminalNode {
    unsigned int sz; // current number of KV pairs
    static const unsigned int cap = CAP; // max number of KV pairs
    ...
};
```

Both default definition fields have type unsigned int which implicitly constrains them to be non-negative as that would reflect a physically impossible node state. The cap definition field is also of type static const which is congruent with the fixed data capacity specified in parameter 9. A constraint that follows from the default definition fields is that the node size $t\rightarrow sz$ cannot be greater than the node capacity $t\rightarrow cap$. Therefore, we add it to the constraint set termInv:

$$termInv \cup \{t\rightarrow sz \leq t\rightarrow cap\}$$
3.4 Utilization and Navigation (5, 22, 23)

The parameter 5. *Utilization* defines the minimum amount of empty data space allowed in the node while the parameters 22. *Sibling Node Links* and 23. *Node Zone Map Filter* define the metadata that should be stored to aid in node navigation. Parameter 22 decides the direction of adjacent sibling node pointers, if any, which enable ordered iteration over the sibling nodes that are scattered randomly throughout memory. Parameter 23 determines whether to keep a zone map of the node—the minimum and/or maximum key contained within the node—which can be used to quickly determine if the node does not contain a particular key-value pair without having to access its data.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Settings</th>
<th>Definition Fields</th>
<th>termInv</th>
</tr>
</thead>
<tbody>
<tr>
<td>5. <em>Utilization</em></td>
<td>≥ 50%</td>
<td></td>
<td>t-&gt;sz ≥ t-&gt;cap / 2</td>
</tr>
<tr>
<td>22. <em>Sibling Node Links</em></td>
<td>NEXT / BOTH</td>
<td>TerminalNode* nxt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PREV / BOTH</td>
<td>TerminalNode* prv;</td>
<td></td>
</tr>
<tr>
<td>23. <em>Node Zone Map Filter</em></td>
<td>MIN / BOTH</td>
<td>int kmin;</td>
<td>t-&gt;kmin = min(K)</td>
</tr>
<tr>
<td></td>
<td>MAX / BOTH</td>
<td>int kmax;</td>
<td>t-&gt;kmax = max(K)</td>
</tr>
</tbody>
</table>

Table 3.2: Terminal Node Utilization and Navigation Definition Field and Constraint Generation Table

Table 3.2 above shows the relevant definition fields and constraints that should be added to the node definition and *termInv* respectively for the different settings listed. We see that the setting of ≥ 50% for 5. *Utilization* naturally leads to the logical constraint that the node
data size must at least be half of the node’s data capacity. For 22. Sibling Node Links, we insert `TerminalNode*` pointer definition fields to represent adjacent sibling node pointers in either direction depending on the given setting. Notice that there is no accompanying logical constraint on the physical state of these pointer definition fields in this generation table. This underspecificity is intentional because the correctness of the `TerminalNode*` pointer definition fields in all the terminal nodes is specified as part of the parent non-terminal node specification. Lastly, for 23. Node Zone Map Filter, recall that $K$ is a logical ordered list of all the keys in the terminal node. Hence, if the node maintains a minimum or maximum key value definition field, the value of that field is the corresponding minimum and maximum value in $K$.

3.5 Key-Value Layout and Ordering (1, 2, 3, 4, 10)

We now turn our attention to specifying the final and most important aspect of the node state - the physical layout and ordering of the key-value data contained within the terminal node. For simplicity, we set both parameters 1. Key Retention and 2. Value Retention to YES by default, which means that the data exists as key-value pairs. This assumption can easily be loosened in the generated specifications to accommodate pure key or value retention. 4. Intra-Node Access, which determines how the node’s data or sub-blocks can be accessed, is also set to DIRECT by default since we expect terminal nodes to be able to access all of the
node data directly. We also assume that the data is fully contained within the node which gives rise to the use of C-style arrays for the key-value definition fields. Therefore, we shall focus on the generative output for the different setting combinations of 3. Key-Value Layout and 10. Key-Fence Partitioning as shown in Table 3.3 and with reference to Figure 3.2 for the definitions of helper predicates used.

Observe from Table 3.3 that there are two possible sets of definition fields that can be generated and that this is determined solely by the setting of 3. Key-Value Layout. A columnar layout means that keys and values are stored in separate segments of memory. Within their respective chunks of memory, keys and values are stored in contiguous order. This thus necessitates two data arrays, each of length cap, in the node. On the other hand, keys and values are stored in the same contiguous segment of memory for both row-wise and row-groups layouts. For such layouts, there need only be a single data array of size cap*2 in the node.

```java
class TerminalNode {
  ...
  int keys[cap];
  int vals[cap];
  /* OR */
  int kvps[cap*2];
  ...
};
```
We see in Figure 3.1 that the difference between row-wise and row-groups layouts then lies in the size of the key-value groups. Within each key-value group, all the keys are stored contiguously before all the values. The relative order amongst the keys and values is preserved such that the $i$-th key and $i$-th value make up the $i$-th key-value pair. Hence, the row-wise layout is simply a special case of the row-groups layout where the group size $\text{GSIZE} = 2$.

The differences between these 3 key-value layouts are expressed in termInv by using 3 different predicates that are defined in Figure 3.2. The predicates list, pairList, and groupedList specify that a given range of memory contains keys and values that are arranged in columnar, row-wise and row-groups layouts respectively. They also specify $K$ and $V$ which are logical ordered lists that respectively contain the keys and values within that same range of memory. Note that $K$ and $V$ are ordered according to increasing physical memory address order and are aligned such that the $i$-th elements in $K$ and $V$ are the key and value of
the $i$-th key-value pair. These logical lists are useful as they decouple ordering constraints from constraints on the physical layout. For instance, the sorted setting for 10. Key-Fence Partitioning means that data is inserted into the node in such a way that preserves sorted order on the keys. This sortedness is easily specified for any of the key-value layouts by bolting on the same sorted predicate over $K$ as follows:

$$termInv \cup \{\text{sorted}(0, t\rightarrow sz, K)\}$$

Similarly, we see that the different settings of 10. Key-Fence Partitioning can be specified irrespective of 3. Key-Value Layout by simply plugging in the right arguments to the corresponding key-value layout predicate. In a forward partitioned node, data is appended starting from the smallest memory address of the data array(s) in increasing memory address order. In contrast, data is appended starting from the last memory address in decreasing memory address order for backward partitioned nodes. A node with sorted partitioning is forward partitioned but with data in sorted order. Therefore, since we know the number of key-value pairs in the node ($sz$) and the maximum number of key-value pairs ($cap$), we can apply the key-value layout predicates over the appropriate segments of memory. The remaining segment of memory, which must have no key-value pairs, is then specified to contain null values by the isNull predicate. Lastly, notice that $sz$ and $cap$ are multiplied by 2 for pairList and groupedList because they are counted in terms of key-value pairs and the kvps array contains both keys and values amongst its $cap*2$ elements.
<table>
<thead>
<tr>
<th>10. Key-Fence Partitioning</th>
<th>Definition Fields</th>
<th>termInv</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>COLUMNAR</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FORWARD</strong></td>
<td>int keys[cap];</td>
<td>(list(&amp;t-&gt;keys), 0, t-&gt;sz, K) ∧ isNull(&amp;t-&gt;keys), t-&gt;sz, t-&gt;cap) * (list(&amp;t-&gt;vals), 0, t-&gt;sz, V) ∧ isNull(&amp;t-&gt;vals), t-&gt;sz, t-&gt;cap)</td>
</tr>
<tr>
<td><strong>BACKWARD</strong></td>
<td>int vals[cap];</td>
<td>(list(&amp;t-&gt;keys), t-&gt;cap - t-&gt;sz, t-&gt;cap, K) ∧ isNull(&amp;t-&gt;keys), 0, t-&gt;cap - t-&gt;sz) * (list(&amp;t-&gt;vals), t-&gt;cap - t-&gt;sz, t-&gt;cap, V) ∧ isNull(&amp;t-&gt;vals), 0, t-&gt;cap - t-&gt;sz)</td>
</tr>
<tr>
<td><strong>SORTED</strong></td>
<td></td>
<td>(list(&amp;t-&gt;keys), 0, t-&gt;sz, K) ∧ sorted(0, t-&gt;sz, K) ∧ isNull(&amp;t-&gt;keys), t-&gt;sz, t-&gt;cap) * (list(&amp;t-&gt;vals), 0, t-&gt;sz, V) ∧ isNull(&amp;t-&gt;vals), t-&gt;sz, t-&gt;cap)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3. Key-Value Layout</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ROW-WISE</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FORWARD</strong></td>
<td>int kvps[cap*2];</td>
<td>pairList(&amp;t-&gt;kvps), 0, 2 × t-&gt;sz, K, V) ∧ isNull(&amp;t-&gt;kvps), 2 × t-&gt;sz, 2 × t-&gt;cap)</td>
</tr>
<tr>
<td><strong>BACKWARD</strong></td>
<td></td>
<td>pairList(&amp;t-&gt;kvps), 2 × (t-&gt;cap - t-&gt;sz), 2 × t-&gt;cap, V, K) ∧ isNull(&amp;t-&gt;kvps), 0, 2 × (t-&gt;cap - t-&gt;sz)</td>
</tr>
<tr>
<td><strong>SORTED</strong></td>
<td>int kvps[cap*2];</td>
<td>pairList(&amp;t-&gt;kvps), 0, 2 × t-&gt;sz, K, V) ∧ sorted(0, t-&gt;sz, K) ∧ isNull(&amp;t-&gt;kvps), 2 × t-&gt;sz, 2 × t-&gt;cap)</td>
</tr>
</tbody>
</table>

| **ROW-GROUPS(GSIZE)**|                   |         |
| **FORWARD**         | int kvps[cap*2];  | groupedList(&t->kvps), 0, 2 × t->sz, GSIZE, K, V) ∧ isNull(&t->kvps), 2 × t->sz, 2 × t->cap) |
| **BACKWARD**        |                   | groupedList(&t->kvps), 2 × (t->cap - t->sz), 2 × t->cap, GSIZE, V, K) ∧ isNull(&t->kvps), 0, 2 × (t->cap - t->sz) |
| **SORTED**          |                   | groupedList(&t->kvps), 0, 2 × t->sz, GSIZE, K, V) ∧ sorted(0, t->sz, K) ∧ isNull(&t->kvps), 2 × t->sz, 2 × t->cap) |

*Table 3.3: Terminal Node Key-Value Definition Field and Invariant Constraint Generation Table*
isNull(a, i, n) ≜ (i = n ∧ emp) ∨
   (i < n ∧ ([a + i, 1] → [null] *isNull(a, i + 1, n)))

sorted(i, n, L) ≜ (i = n - 1) ∨
   (i < n - 1 ∧ x ≤ y ∧ L[i] = x ∧ L[i + 1] = y ∧ sorted(i + 1, n, L))

list(a, i, n, L) ≜ (i = n ∧ L = [[ ]] ∧ emp) ∨
   (i < n ∧ L = [[x]] || L_1 ∧ ([a + i, 1] → [x] * list(a, i + 1, n, L_1)))

pairList(a, i, n, E, O) ≜ (i = n ∧ E = [[ ]] ∧ O = [[ ]] ∧ emp) ∨
   (i < n ∧ E = [[x]] || E_1 ∧ O = [[y]] || O_1 ∧ ([a + i, 2] → [x, y] * pairList(a, i + 2, n, E_1, O_1))

groupedList(a, i, n, s, E, O) ≜ (n - i ≤ s ∧ E = E_1 ∧ O = O_1 ∧ (list(a + i, 0, (n - i)/2, E_1) * list(a + i, (n - i)/2, n - i, O_1)) ∨
   (n - i > s ∧ E = E_1 || E_2 ∧ O = O_1 || O_2 ∧ (list(a + i, 0, s/2, E_1) *
              list(a + i, s/2, s, O_1) * groupedList(a, i + s, n, s, E_2, O_2))))

Figure 3.2: Helper Predicate Definitions for Key-Value Layout and Ordering Constraints
As implied by their name, non-terminal nodes refer to nodes that are described by non-terminal elements and hence have children nodes or sub-blocks. Their primary function is to partition the data over different sub-block layouts, thereby describing how child nodes can be navigated and how key-value pairs are distributed across the data structure. By com-
posing together various configurations of non-terminal nodes, a data structure can thus partition different segments of the data space in a tailored fashion. A non-terminal node may also aggregate sub-block filters which eliminate accesses to sub-blocks without the relevant data during data structure operations. In this chapter, we demonstrate how non-terminal node definitions and invariant specifications can be procedurally generated from the parameters of a non-terminal element.

4.1 Non-Terminal Node Parameters

Table 4.1 below shows the parameters and settings that directly affect the generated non-terminal node definitions and specifications. For non-terminal nodes, we make the simplifying assumption that they do not contain data and thus only terminal nodes store data. Hence, 1. Key Retention and 2. Value Retention are set to NO by default. In addition, the backward setting for 10. Key-Fence Partitioning is omitted to maintain clarity of presentation by avoiding a multiplicity of interacting setting combinations. This does not affect the generalizability of the specifications as the backward and forward cases only differ in the memory location of the initialized sub-blocks. We leave some brief comments wherever the backward case is relevant to aid the reader in understanding how the specification generation can be easily extended to account for this setting.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Retention</td>
<td>NO</td>
</tr>
<tr>
<td>Value Retention</td>
<td>NO</td>
</tr>
<tr>
<td>Key-Value Layout</td>
<td>N.A.</td>
</tr>
<tr>
<td>Intra-Node Access</td>
<td>DIRECT, HEAD, TAIL</td>
</tr>
<tr>
<td>Utilization</td>
<td>NONE</td>
</tr>
<tr>
<td>Bloom Filters</td>
<td>OFF</td>
</tr>
<tr>
<td>Zone Map Filters</td>
<td>MIN, MAX, BOTH, OFF</td>
</tr>
<tr>
<td>Filters Memory Layout</td>
<td>CONSOLIDATED, SCATTERED</td>
</tr>
<tr>
<td>Fanout / Radix</td>
<td>FIXED(FAN), UNLIMITED</td>
</tr>
<tr>
<td>Key-Fence Partitioning</td>
<td>FORWARD, SORTED</td>
</tr>
<tr>
<td>Sub-Block Capacity</td>
<td>FIXED(SBCAP), UNRESTRICTED, BALANCED</td>
</tr>
<tr>
<td>Immediate Node Links</td>
<td>NEXT, PREV, BOTH, NONE</td>
</tr>
<tr>
<td>Skip Node Links</td>
<td>NONE</td>
</tr>
<tr>
<td>Area Links</td>
<td>NONE</td>
</tr>
<tr>
<td>Sub-Block Physical Location</td>
<td>INLINE, POINTED</td>
</tr>
<tr>
<td>Sub-Block Physical Layout</td>
<td>BFS, SCATTER</td>
</tr>
<tr>
<td>Sub-Blocks Homogeneous</td>
<td>TRUE</td>
</tr>
<tr>
<td>Sub-Block Consolidation</td>
<td>FALSE</td>
</tr>
<tr>
<td>Sub-Block Instantiation</td>
<td>LAZY, EAGER</td>
</tr>
<tr>
<td>Sub-Block Links Layout</td>
<td>CONSOLIDATE, SCATTER</td>
</tr>
<tr>
<td>Recursion</td>
<td>NO</td>
</tr>
<tr>
<td>Sibling Node Links</td>
<td>NEXT, PREV, BOTH, NO</td>
</tr>
<tr>
<td>Node Zone Map Filter</td>
<td>MIN, MAX, BOTH, NO</td>
</tr>
</tbody>
</table>

Table 4.1: Relevant Parameters and Settings for Generating Non-Terminal Node Definitions and Invariant Specifications.
4.2 Non-Terminal Node Invariant Specification

The non-terminal node invariant specification is defined as follows:

\[
\text{nodeInv}_n(t, K, V) \triangleq \text{subBlocks}_n(t, \text{firstChild}, 0, t\rightarrow\text{sbinit}, K, V, C) \land \bigwedge_{c \in NTCons} c
\]

\[
\text{subBlocks}_n(t, c, i, n, K, V, C) \triangleq (i = n \land K = \{\}\land V = \{\}\land C = 0 \land \text{emp}) \lor
\]

\[
(i < n \land K = \{K_1\} \parallel K_2 \land V = \{V_1\} \parallel V_2
\]

\[
\land C = c\rightarrow\text{cap} + C_2 \land \bigwedge_{c \in SBCons} c
\]

\[
\land (\text{nodeInv}_{child}(c, K_1, V_1) \ast
\]

\[
\text{subBlocks}_n(t, \text{nextChild}, i + 1, n, K_2, V_2, C_2))
\]

In contrast to the terminal node invariant specification which was just a single set of constraints, the non-terminal node invariant specification comprises the constraint set \(NTCons\), which holds general constraints on the non-terminal node state, as well as the \(\text{subBlocks}\) predicate which ranges over the node’s initialized sub-blocks. This structure facilitates the specification of sub-block correctness because it allows constraints to be asserted at the sub-block granularity as part of the constraint set \(SBCons\) within \(\text{subBlocks}\). In particular, relationships between adjacent sub-blocks are easily specified as a result.

The predicate \(\text{nodeInv}_{child}\) represents the corresponding child node invariant specification that is generated from the parameters of the child element in the Data Calculator data structure design. This can be a terminal node invariant as generated in Chapter 3 or it could specify a different type of non-terminal node invariant.
Another distinct feature of this node invariant specification is the use of generation variables. \textit{firstChild} and \textit{nextChild} are logical variables that take on different values depending on the node’s sub-block definition fields, which are in turn determined by various parameter settings. The variable \textit{firstChild} is the memory location of the first initialized sub-block, while \textit{nextChild} is the memory location of the \( i + 1 \)-th sub-block.

Similar to terminal nodes, the variables \( K \) and \( V \) are respectively defined by \textit{subBlocks} as lists containing the keys and values under the node. However, unlike terminal nodes, they are not flat lists but rather nested lists that are split and ordered according to sub-block. If the child nodes are non-terminal, these list elements will be nested lists too. Consequently, this recursively defined hierarchy of lists mirrors the partitioning by non-terminal nodes, which enables precise reasoning about data movement caused by data structure operations. We expound on this further in Chapter 5. Additionally, \textit{subBlocks} defines another variable \( C \) which is simply the sum of all the sub-block data capacities.

Lastly, we acknowledge that this non-terminal node invariant specification assumes for-ward or sorted 10. Key-Fence Partitioning as mentioned previously. Hence, the \textit{subBlocks} predicate is hard-coded to range over the first \( t->sbinit \) sub-blocks, where \( t->sbinit \) is the number of the node’s sub-blocks that are initialized. For the backward case, \textit{subBlocks} should instead be asserted over the last \( t->sbinit \) sub-blocks.
4.3 Sub-Block Metadata (9, 11, 19)

The default metadata that needs to be stored in a non-terminal node is informed by two parameters. Firstly, 19. Sub-Block Instantiation dictates whether all sub-blocks should be initialized (eager) or only non-empty sub-blocks should be initialized (lazy). This parameter implies that there needs to be a metadata value that keeps track of the number of initialized sub-blocks. Secondly, 11. Sub-Block Capacity designates the maximum number of key-value pairs that a non-terminal node’s sub-block can have underneath it. The sub-block capacity can be fixed at \texttt{SBCAP}, unrestricted, or balanced which means that the data is equally divided amongst the sub-blocks. In any case, a non-terminal node’s sub-blocks should store capacity metadata so that the correctness of sub-block capacities can be specified. Since a non-terminal node may itself be a sub-block of another non-terminal node, it must have a data capacity definition field. Note that terminal nodes already have such a definition field by default as explained in Section 3.3. Therefore, the default non-terminal node metadata definition fields are as follows:

```c
class NonTerminalNode {
    unsigned int sbinit; // current number of initialized sub-blocks
    unsigned int cap; // total data capacity of all child nodes
    ...
};
```
Notice that both definition fields are of type `unsigned int` which implicitly constrains them to be non-negative as it is physically impossible to have a data capacity or number of initialized sub-blocks that is less than zero. Neither is declared as `const` since `sbinit` will vary if the maximum number of sub-blocks changes or if the sub-blocks are instantiated lazily, while `cap` is dynamic if a descendant node has `UNRESTRICTED` capacity.

Recall that $C$ from the non-terminal node invariant specification is recursively defined as the sum of all child node data capacities. Hence, it immediately follows that:

$$ NTCons \cup \{t\rightarrow cap = C\} $$

For `sbinit`, we constrain its value with respect to the node fanout (\(fan\)) — the maximum number of sub-blocks — which is defined by 9. Fanout / Radix. The node fanout is either `FIXED` at some value `FAN` or is `UNLIMITED` and varies without restriction. The resulting definition fields and constraints to be generated are summarized in Table 4.2 below.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Settings</th>
<th>Definition Fields</th>
<th>NTCons</th>
</tr>
</thead>
<tbody>
<tr>
<td>9. Fanout / Radix</td>
<td>FIXED(<code>FAN</code>)</td>
<td><code>static const unsigned int fan = FAN;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UNLIMITED</td>
<td><code>unsigned int fan;</code></td>
<td></td>
</tr>
<tr>
<td>19. Sub-Block Instantiation</td>
<td>LAZY</td>
<td><code>t\rightarrow sbinit \leq t\rightarrow fan</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EAGER</td>
<td><code>t\rightarrow sbinit = t\rightarrow fan</code></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Non-Terminal Node Metadata Definition Field and Constraint Generation Table
We see that the node fanout is also an unsigned int definition field. This is because a non-terminal node without any sub-blocks would be terminal since it has no children nodes which is a contradiction. Under eager 19. Sub-Block Instantiation, \( sbinit \) is equal to \( \text{fan} \) as all sub-blocks are initialized. Conversely, empty sub-blocks are not initialized with lazy instantiation and thus \( sbinit \) can be less than or at most equal to \( \text{fan} \).

4.4 Navigation (22, 23)

Just like terminal nodes, non-terminal nodes that are sub-blocks may store zone map filters and pointers to sibling nodes. The definition fields and constraints for such navigation information are given in Table 4.3 which is identical to that for terminal nodes. The one exception is that the sibling pointer type \( \text{NonTerminalNode}^* \) can be named differently as a data structure can consist of several different non-terminal node classes. In contrast, there can only be a single terminal node class because 17. Sub-Blocks Homogeneous is set to TRUE by default for all nodes. For further explanation of the table, please refer to Section 3.4.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Settings</th>
<th>Definition Fields</th>
<th>NTCons</th>
</tr>
</thead>
<tbody>
<tr>
<td>22. Sibling Node Links</td>
<td>NEXT / BOTH</td>
<td>NonTerminalNode* nxt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PREV / BOTH</td>
<td>NonTerminalNode* prv;</td>
<td></td>
</tr>
<tr>
<td>23. Node Zone Map Filter</td>
<td>MIN / BOTH</td>
<td>int kmin;</td>
<td>( t \rightarrow kmin = \min(K) )</td>
</tr>
<tr>
<td></td>
<td>MAX / BOTH</td>
<td>int kmax;</td>
<td>( t \rightarrow kmax = \max(K) )</td>
</tr>
</tbody>
</table>

Table 4.3: Non-Terminal Node Navigation Definition Field and Constraint Generation Table
4.5 Zone Map Filters (7, 8, 9, 10, 19)

Recall that a zone map is the minimum and/or maximum key that is present under a node and that it can be used to filter out unnecessary sub-block accesses during data structure operations. A non-terminal node may store zone maps within each sub-block under a scattered 8. Filters Memory Layout, or it may have a consolidated 8. Filters Memory Layout where sub-block zone maps are stored together in a buffer. As shown in the previous section, scattered zone maps are easily specified in the child node invariant specification with the 23. Node Zone Map Filter parameter. Therefore, in this section, we need only account for the consolidated case.

Before we can consider the appropriate definition fields and constraints, we must first address the fact that the physical layout of the zone map filters is not fully described by the parameters. In Section 3.5, we saw that a terminal node either stores keys and values in separate buffers or it has a single key-value buffer that stores data in different layouts based on 3. Key-Value Layout. Similarly, for non-terminal nodes, we have the equivalent choice of storing the consolidated minimum and maximum zone maps separately or together in a variety of arrangements.

Another aspect of the zone map layout that is not parameterized is the relative location of the zone map buffers in memory — they either reside within the node or elsewhere in memory. This is similar to the information given by 15. Sub-Block Physical Location which
we expand on in the next section. The parameter 9. Fanout / Radix partially determines
the relative location of the zone map buffers because an UNLIMITED fanout necessitates
the resizing of zone map buffers to accommodate the changing number of sub-blocks. Such
dynamic length buffers must exist outside the node as inlined array members of unknown
bound are incomplete types according to the C++ specification. However, for FIXED node
fanouts, both relative zone map buffer locations are valid.

In addition, the parameters do not provide any information about the appropriate size
of the zone map buffers when 19. Sub-Block Instantiation is LAZY. Since zone map buffers
outside the node can be resized or reallocated as needed, the buffer size can vary between
the number of initialized sub-block zone maps (sbinit) and the maximum number of
zone maps (fan). Note that for EAGER 19. Sub-Block Instantiation the node fanout is equal
to the number of initialized sub-blocks and hence there is no ambiguity about the length of
the zone map buffers.

For the purposes of this section, we assume separate minimum and maximum zone map
buffers. For non-terminal nodes with FIXED fanout, we assume that the zone map buffers
are fully contained within the node and have fixed length equal to the node fanout. Con-
versely, we assume that non-fixed length buffers outside nodes with UNLIMITED fanout
are used to store the zone maps and that the buffers are resized to be equal to the changing
node fanout.
In Table 4.4, we see that the zone map buffers for non-terminal nodes with fixed fanout are specified as arrays with length equal to the node fanout. This is valid because the array length \( \text{fan} \) is the constant known value of \( \text{FAN} \) when the node fanout is fixed. For nodes with unlimited fanout, the appropriate zone map definition field type is a vector since it is implemented as a pointer to a heap-allocated buffer. Unlike arrays, the length of the vector is not specified as part of the definition field and so it is constrained to be equal to the node fanout in \( \text{NTCons} \):

\[
\text{NTCons} \cup \{ \text{t} \rightarrow \text{zmin.size} = \text{t} \rightarrow \text{fan} \} \cup \{ \text{t} \rightarrow \text{zmax.size} = \text{t} \rightarrow \text{fan} \}
\]

To specify the correctness of the min and max zone map buffers, we need to stipulate that: (1) they contain the right minimum or maximum keys of the initialized sub-blocks in the right positions and that (2) the uninitialized sub-block zone maps are null valued.
The first condition is specified by imposing the constraint:

\[ \text{SBCons} \cup \{ t_{->zmin[i]} = \min(K_1) \} \cup \{ t_{->zmax[i]} = \max(K_1) \} \]

This constraint is applied to every initialized sub-block as part of \text{SBCons} and asserts that the \(i\)-th zone map buffer value is equal to the minimum or maximum key underneath the \(i\)-th initialized child node.

The second condition is satisfied by simply declaring the isNull predicate over the zone map buffer values corresponding to the uninitialized sub-blocks:

\[ \text{NTCons} \cup \{ \text{isNull}(t_{->zmin}, t_{->sbinit}, t_{->fan}) \} \cup \{ \text{isNull}(t_{->zmax}, t_{->sbinit}, t_{->fan}) \} \]

We note that this constraint assumes forward key-fence partitioning as the zone maps are taken to be the first \(t_{->sbinit}\) values in the zone map buffer and hence isNull is asserted on the latter portion of the zone map buffer. For backward key partitioning, subBlocks will range over the last \(t_{->sbinit}\) sub-blocks and so isNull is applicable only for the first \(t_{->fan} - t_{->sbinit}\) zone map buffer values.

4.6 Sub-Block Layout and Link Correctness (4, 9, 10, 12, 15, 16, 19, 20)

Rather than evaluating all possible parameter combinations, we instead apply our knowledge of the C++ abstract machine model to derive that there are only five possible sub-block definition field types. Intuitively, if the parent non-terminal node is able to access
all the sub-blocks directly (4. Intra-Node Access = DIRECT), then it must have access to a buffer that contains all the child nodes or pointers to them. This buffer can reside within the parent non-terminal node (Child sb[fan] / Child* sb[fan]) or it exists elsewhere in memory and is accessed via a pointer stored in the node (vector<Child> sb / vector<Child*> sb). Alternatively, if only the first or last sub-block can be accessed by the parent node (4. Intra-Node Access = head / tail), then the node just contains a single pointer to that sub-block (Child* sbhd / Child* sbtl). Consequently, the only way the other sub-blocks can be accessed must be through 12. Immediate Sub-Block Links which are pointers that connect sibling nodes.

Table 4.5 details the corresponding parameter settings and variable values for each of the aforementioned definition fields, along with the relevant constraints on the node state. From the previous section, we know that the inlined sub-block arrays are only valid with fixed 9. Fanout / Radix because the node fanout will be a constant known value. On the other hand, the sub-block vectors can have fixed or dynamic length, since the buffers being pointed to are outside the node and thus may be resized or reallocated accordingly. Similar to zone map buffers, under lazy 19. Sub-Block Instantiation, we note that the parameters do not precisely specify the sub-block vector size which can vary between the number of initialized sub-blocks (sbinit) and the maximum number of sub-blocks (fan). In this section, we assume that the vector sizes are tied to the node fanout.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Child sb[fan]; uninit(t-&gt;sb, t-&gt;sbinit, t-&gt;fan)</td>
<td>t-&gt;sb              &amp; (t-&gt;sb[i+1])</td>
<td>DIRECT</td>
<td>FIXED</td>
<td>NONE</td>
<td>INLINE</td>
<td>BFS</td>
<td>N.A.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Child* sb[fan];</td>
<td>t-&gt;sb[0] t-&gt;sb[i+1]</td>
<td>DIRECT</td>
<td>FIXED</td>
<td>NONE</td>
<td>POINTED</td>
<td>SCATTER</td>
<td>CONSOLIDATE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vector&lt;Child&gt; sb; t-&gt;sb.size = t-&gt;fan \ uninit(t-&gt;sb, t-&gt;sbinit, t-&gt;fan)</td>
<td>t-&gt;sb              &amp; (t-&gt;sb[i+1])</td>
<td>DIRECT</td>
<td>FIXED / UNLIMITED</td>
<td>NONE</td>
<td>POINTED</td>
<td>BFS</td>
<td>N.A.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vector&lt;Child*&gt; sb; t-&gt;sb[0] t-&gt;sb[i+1]</td>
<td>DIRECT</td>
<td>FIXED / UNLIMITED</td>
<td>NONE</td>
<td>POINTED</td>
<td>SCATTER</td>
<td>CONSOLIDATE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Child* sbhd;</td>
<td>t-&gt;sbhd c-&gt;nxt</td>
<td>HEAD</td>
<td>FIXED / UNLIMITED</td>
<td>NEXT / BOTH</td>
<td>POINTED</td>
<td>SCATTER</td>
<td>SCATTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Child* sbtl;</td>
<td>t-&gt;sbtl c-&gt;prv</td>
<td>TAIL</td>
<td>FIXED / UNLIMITED</td>
<td>PREV / BOTH</td>
<td>POINTED</td>
<td>SCATTER</td>
<td>SCATTER</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Generation Table for Valid Non-Terminal Node Sub-Block Layouts
Equality between the vector size and node fanout is thus enforced with the constraint:

$$NTCons \cup \{t->sb.size = t->fan\}$$

One implication of this assumption is that part of the sub-block vector will contain uninitialized sub-blocks or null sub-block pointers. In either case, that particular segment of memory should be null valued in order to be correct. This applies to the inlined sub-block arrays as well since they have fixed size equal to the node fanout. As such, we assert that:

$$NTCons \cup \{uninit(t->sb, t->sbinit, t->fan)\}$$

where uninit is defined as:

$$uninit(s, c, i, n) \triangleq (i = n \land emp) \lor$$

$$\quad (i < n \land (isNull(s, c, nextChild) \land uninit(s, nextChild, i + 1, n)))$$

The declaration of uninit over the latter portion of the sub-block buffers points to the assumption of forward 10. Key-Fence Partitioning. For backward key partitioning, the initialized sub-blocks are the last t->sbinit sub-blocks, and so uninit should instead be specified for the first t->fan - t->sbinit sub-blocks in the respective buffers. Note that the variable nextChild used in uninit is the memory location of the $i + 1$-th sub-block, which is t->sb[i+1] if the sub-block buffer stores pointers or is &t->sb[i+1] if the buffer contains the actual sub-blocks. Similarly, firstChild — the memory location of the
first sub-block—differs depending on whether the contents of the buffer are actual sub-blocks ($t\rightarrow sb$) or pointers to them ($t\rightarrow sb[0]$).

Given that an arbitrary number of sub-blocks can be chained together with pointers, the fanout or maximum number of linked sub-blocks can be fixed or dynamic too. However, unlike the array and vector definition fields, only the initialized linked sub-blocks are specified as shown in Table 4.6 below. This means that the uninitialized linked sub-blocks are not allocated and are implicitly null as described by 19. Sub-Block Instantiation. We acknowledge that a data structure may instead allocate segments of null-valued memory for the uninitialized linked sub-blocks, but we omit this extra parameterization in order to maintain parity with the parameters.

<table>
<thead>
<tr>
<th>4. Intra-Node Access</th>
<th>12. Immediate Node Links</th>
<th>SBCons</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HEAD</strong></td>
<td><strong>NEXT</strong></td>
<td>$((i = n - 1 \land c\rightarrow nxt = null) \lor (i &lt; n - 1))$</td>
</tr>
</tbody>
</table>
|                      | **BOTH**                 | $((i = 0 \land c\rightarrow prv = null) \lor i > 0) \land$
|                      |                           | $((i < n - 1 \land c\rightarrow nxt\rightarrow prv = c) \lor (i = n - 1 \land c\rightarrow nxt = null))$ |
| **TAIL**             | **PREV**                 | $(i = n - 1 \land c\rightarrow prv = null) \lor (i < n - 1)$ |
|                      | **BOTH**                 | $((i = 0 \land c\rightarrow nxt = null) \lor i > 0) \land$
|                      |                           | $((i < n - 1 \land c\rightarrow prv\rightarrow nxt = c) \lor (i = n - 1 \land c\rightarrow prv = null))$ |

Table 4.6: Non-Terminal Node Sub-Block Link Correctness Constraint Generation Table
The sub-block link correctness relies on the child node invariant \( \text{nodeInv}_{\text{child}}(c, K_1, V_1) \) in the non-terminal node invariant specification to specify that there is a valid sub-block at the memory location of the \( i \)-th sub-block \( (c) \). Since \( c \) is the \texttt{nxt} or \texttt{prv} value of the \( i-1 \)-th sub-block as defined by \( \text{nextChild} \) for \( \texttt{sbhd/sbtl} \), this recursively specifies the correctness of the linked sub-blocks.

Table 4.6 details the additional constraints needed to fully specify the sub-block link correctness. The last sub-block in either direction must not point to anything and hence has a \texttt{nxt} or \texttt{prv} value of \texttt{null} for \texttt{head} and \texttt{tail} respectively. For both kinds of \texttt{12. Immediate Node Links}, we make sure to assert that the \texttt{nxt} and \texttt{prv} values of adjacent sub-blocks corroborate with each other too.

Conversely, non-terminal nodes with \texttt{direct} access to all their sub-blocks have no need
for 12. **Immediate Node Links**. However, some of the definition fields store buffers of pointers \((\text{Child* sb}[\text{fan}])\) and thus 20. **Sub-Block Links Layout** is set to **CONSOLIDATE** in comparison to **SCATTER** for the linked sub-blocks.

We now address the two remaining parameters 15. **Sub-Block Physical Location** and 16. **Sub-Block Physical Layout**. If a non-terminal node’s sub-blocks are fully contained within the node, it has **INLINE** 15. **Sub-Block Physical Location**. Otherwise, the node has **POINTED** 15. **Sub-Block Physical Location** which indicates that sub-blocks exist at arbitrary memory locations outside the node and thus must be referenced by pointers. Hence, this parameter clearly differentiates \(\text{Child* sb}[\text{fan}]\) from the other definition fields. Child nodes that are stored contiguously within a buffer are described by a **BFS** 16. **Sub-Block Physical Layout**, while child nodes that are scattered across memory are denoted by a **SCATTER** 16. **Sub-Block Physical Layout**. This parameter thus distinguishes a buffer of child nodes from a buffer of pointers to child nodes or the linked sub-blocks.

Here, we observe that the parameter settings that match \(\text{Child* sb}[\text{fan}]\) are valid for generating \(\text{vector<Child*> sb}\) as well. This implies that the Data Calculator parameters do not have the appropriate settings to differentiate between these two definition fields.
4.7 Sub-Block Capacity (11)

In Table 4.7, we see that $SBCons$ affords us concise constraints. We can state that each sub-block has fixed capacity $SBCAP$ or that sub-block capacities are balanced by asserting capacity equality between neighbouring sub-blocks. Notice that unrestricted is conspicuously absent as there is no need to constrain the sub-block capacities in that case.

<table>
<thead>
<tr>
<th>11. Sub-Block Capacity</th>
<th>$SBCons$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed($SBCAP$)</td>
<td>$c-&gt;cap = SBCAP$</td>
</tr>
<tr>
<td>Balanced</td>
<td>$(i &lt; n-1 \land c-&gt;cap = nextChild-&gt;cap) \lor i = n-1$</td>
</tr>
</tbody>
</table>

Table 4.7: Non-Terminal Node Sub-Block Capacity Constraint Generation Table

4.8 Sorted Invariant (10)

Lastly, for non-terminal nodes which maintain sorted key-partitioning, we simply assert that the flattened version of $K$ is sorted. Further correctness for 10. Key-Fence Partitioning is specified in Chapter 5 when describing changes to the logical data structure state.

<table>
<thead>
<tr>
<th>10. Key-Fence Partitioning</th>
<th>$NTCons$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted</td>
<td>$\text{sorted}(0, flatten(K), flatten(K))$</td>
</tr>
</tbody>
</table>

Table 4.8: Non-Terminal Node Sorted Invariant
Now that we are able to generate node invariant specifications, we complete the functional specifications by presenting the list manipulation predicates for the Get, Delete, and Insert operations. The effect of these operations on the logical data structure state is not fully described by the Data Calculator parameters. Hence, we state any areas of underspeci-
ficity and explain their potential impact on the synthesized implementation.

5.1 Get

The Hoare triple for the Get operation is:

\[
\begin{align*}
\{ \text{nodeInv}_n(t, K, V) \} \\
\text{void get}(\text{int} \, \text{key}, \text{int*} \, \text{vloc}) \\
\{ \text{retrieved}(\text{key}, \text{vloc}, K, V) \land \text{nodeInv}_n(t, K, V) \}
\end{align*}
\]

We define retrieved and the helper predicate zip as follows:

\[
\begin{align*}
\text{retrieved}(k, \text{vloc}, K, V) & \triangleq (\exists v. \, [\text{vloc}, 1] \mapsto [v] \land \text{zip}(0, |K|, \text{flatten}(K), \text{flatten}(V), Z) \land (k, v) \in Z) \lor \\
& (\text{[vloc, 1] \mapsto [null] \land k \not\in \text{flatten}(K)}) \\
\text{zip}(i, n, L_1, L_2, Z) & \triangleq (i = n) \lor \\
& (i < n \land Z = \{ (L_1[i], L_2[i]) \} \cup Z_1 \land \text{zip}(i + 1, n, L_1, L_2, Z_1))
\end{align*}
\]

The function argument int* vloc points to a memory location at which the function either places the value associated with key or a null value if key does not exist in the data structure. Notice that the node invariants in the pre-condition and the post-condition share the same variables K and V because the Get operation should not modify any data structure state. As a reminder, flatten(K) denotes a flat version of K and hence zip is simply used to associate the key-value pairs of the data structure in a multiset.
The specification of retrieved assumes that the data structure only contains unique keys or that a valid function can use any arbitrary tie-breaking measure to decide which value to return if there are multiple keys that match. If desired, an explicit tie-breaking measure can be easily specified by bolting on the appropriate constraint in the key exists case. Alternatively, the function signature can be changed to allow for multiple matching values to be placed at multiple result memory locations.

5.2 Delete

The Hoare Triple for the Delete operation is:

\[
\{\text{nodeInv}_n(t, K_1, V_1)\}
\]

\[
\text{void del(int key)}
\]

\[
\{\text{deleted(key, } K_1, K_2, V_1, V_2) \land \text{nodeInv}_n(t, K_2, V_2)\}\]

We define deleted as follows:

\[
\text{deleted}(k, K_1, K_2, V_1, V_2) \overset{\Delta}{=} (\exists v. \text{zip}(0, |K_1|, \text{flatten}(K_1), \text{flatten}(V_1), Z_1) \land \text{zip}(0, |K_2|, \text{flatten}(K_2), \text{flatten}(V_2), Z_2) \land Z_2 = Z_1 - \{(k, v)\} \lor (k \notin \{\text{flatten}(K_1)\} \land K_1 = K_2 \land V_1 = V_2)
\]

We see that if the key does not exist in the data structure then the key and value lists must remain the same since no data structure state should be modified. If the key does
exist, deleted asserts that the corresponding key-value pair must be removed from the data structure. Note that changes to the nested structure or length of the key and value lists are intentionally left unspecified as this allows for implementations that can split and merge nodes accordingly to manage the empty space in the data structure. Similar to retrieved, constraints can be added to the key exists case of deleted to control what kind of data structure reorganizations are allowed.

The node invariant specification may also influence the behaviour of the Delete operation. As described in Section 3.5, the data in a terminal node must be contiguous and must start from either end of the data array(s). This implicitly specifies that the Delete operation must shift the terminal node data appropriately to maintain this contiguity. Additionally, 

5. Utilization constrains the amount of empty space a node can have. For example, if it is set to $\geq 50\%$, then any data structure reorganization must distribute the data across the nodes in a way that maintains this invariant.

Lastly, we observe that there is no Data Calculator parameter to specify delayed deletes or pseudo-deletes involving the setting of a deleted bit to invalidate the data record without removing it. These are techniques used in modern data systems to mitigate the costs of deletes. Specifying these procedures requires the fundamental design concepts at play to be ascertained and parameterized appropriately.
5.3 Insert

The Hoare Triple for the Insert operation is:

\[
\{ \text{nodeInv}_n(t, K_1, V_1) \}
\]

\[
\text{void insert(int key, int value)}
\]

\[
\{ \text{inserted}_{n}(\text{key, value, } K_1, K_2, V_1, V_2, |K_1|, |K_2|) \land \text{nodeInv}_n(t, K_2, V_2) \}
\]

Unlike retrieved and deleted, inserted is defined differently for non-terminal and terminal elements as well as for different settings of 10. Key-Fence Partitioning. The inserted definition for terminal elements is:

\[
\text{inserted}_n(k, v, K_1, K_2, V_1, V_2, N_1, N_2) \triangleq \text{insPostCond}
\]

<table>
<thead>
<tr>
<th>10. Key-Fence Partitioning</th>
<th>insPostCond</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORWARD</td>
<td>( K_2 = K_1 \parallel {[k]} \land V_2 = V_1 \parallel {[v]} )</td>
</tr>
<tr>
<td>BACKWARD</td>
<td>( K_2 = {[k]} \parallel K_1 \land V_2 = {[v]} \parallel V_1 )</td>
</tr>
<tr>
<td>SORTED</td>
<td>( \text{zip}(0, N_1, K_1, V_1, Z_1) \land \text{zip}(0, N_2, K_2, V_2, Z_2) \land Z_2 = Z_1 \cup \langle k, v \rangle )</td>
</tr>
</tbody>
</table>

Table 5.1: Terminal Element Insert Post-Condition

The forward and backward partitioned terminal nodes simply append the new key-value pair at the appropriate end of the list. For a sorted terminal node, we assert that the data structure contains the same data, but with the addition of the new key-value pair
after the insert. Here, the terminal node invariant specification ensures that the insertion location of the key-value pair maintains the sorted ordering of the keys. Furthermore, the key-value layout constraints of the terminal node invariant ensure the contiguity of the data as well.

For non-terminal elements, \textit{insPostCond} has two distinct cases regardless of the key partitioning —either the number of sub-blocks increases by 1 or it stays the same. If the number of sub-blocks is unchanged, then the key-value pair must be inserted under a particular sub-block which we can recursively specify. The state of all other sub-blocks must be the same. The index of the sub-block that receives the key-value pair is either the first of the last sub-block by definition for \texttt{backward} and \texttt{forward} key partitioning respectively. For \texttt{sorted}, we specify with the predicate \texttt{sortedInsertIndex} that the key-value pair is inserted into the first sub-block that has a max key value greater than or equal to \texttt{key}.

If the number of sub-blocks increases by 1, it means that a new sub-block is created at the end of the list corresponding to the partitioning direction. For \texttt{forward} and \texttt{backward} key partitioning, this new sub-block must contain just the inserted key-value pair and the state of all other sub-blocks must remain the same. However, for a \texttt{sorted} non-terminal node, the key-value pair is inserted in sorted order and thus may not end up in the new sub-block. Hence, we simply assert that the new data structure state contains all previous data and the new key-value pair.
\text{sameExcept}(j, i, n, K_1, K_2, V_1, V_2) \triangleq (i = n) \lor \\
(i < n \land i = j \land \text{sameExcept}(j, i + 1, n, K_1, K_2, V_1, V_2)) \lor \\
(i < n \land i \neq j \land K_1(i) = K_2(i) \land V_1(i) = V_2(i)) \\
\land \text{sameExcept}(j, i + 1, n, K_1, K_2, V_1, V_2)) \\

\text{sortedInsertIndex}(k, i, n, K, S) \triangleq (i = n - 1 \land S = i) \lor \\
(i < n \land k \leq \text{max}(K[i]) \land S = i) \lor \\
(i < n \land k > \text{max}(K[i]) \land S = S_1) \\
\land \text{sortedInsertIndex}(k + 1, n, K_1, K_2, S_1)) \\

\text{insPostCond}_{\text{forward}} \triangleq (N_2 = N_1 + 1 \land \text{flatten}(K_2(\langle N_1 \rangle)) = \{[k] \land \text{flatten}(V_2(\langle N_1 \rangle)) = \{[v] \\
\land K_1 = \text{K}_2(\langle : N_1 \rangle) \land V_1 = V_2(\langle : N_1 \rangle)) \lor \\
(N_2 = N_1 \land \text{sameExcept}(N_2, 0, N_2, K_1, K_2, V_1, V_2) \\
\land \text{inserted}_{\text{child}}(k, v, K_1(\langle N_2 \rangle), K_2(\langle N_2 \rangle), V_1(\langle N_2 \rangle), V_2(\langle N_2 \rangle))) \\

\text{insPostCond}_{\text{backward}} \triangleq (N_2 = N_1 + 1 \land \text{flatten}(K_2(\langle 0 \rangle)) = \{[k] \land \text{flatten}(V_2(\langle 0 \rangle)) = \{[v] \\
\land K_1 = K_2(\langle 1 : \rangle) \land V_1 = V_2(\langle 1 : \rangle)) \lor \\
(N_2 = N_1 \land \text{sameExcept}(0, 0, N_2, K_1, K_2, V_1, V_2) \\
\land \text{inserted}_{\text{child}}(k, v, K_1(\langle 0 \rangle), K_2(\langle 0 \rangle), V_1(\langle 0 \rangle), V_2(\langle 0 \rangle))) \\

\text{insPostCond}_{\text{sorted}} \triangleq (N_2 = N_1 + 1 \land \text{zip}(0, N_1, \text{flatten}(K_1), \text{flatten}(V_1), Z_1) \\
\land \text{zip}(0, N_2, \text{flatten}(K_2), \text{flatten}(V_2), Z_2) \\
\land Z_2 = Z_1 \cup \{\langle k, v \rangle\}) \lor \\
(N_2 = N_1 \land \text{sortedInsertIndex}(k, 0, N_1, K_1, S) \\
\land \text{sameExcept}(S, 0, N_2, K_1, K_2, V_1, V_2) \\
\land \text{inserted}_{\text{child}}(k, v, K_1(\langle S \rangle), K_2(\langle S \rangle), V_1(\langle S \rangle), V_2(\langle S \rangle)))
Conclusion

In this thesis, we demonstrate how data structure specifications for the purposes of data structure synthesis can be procedurally generated from a Data Calculator data structure design. These specifications consist of node definitions for each design element as well as functional specifications for the Get, Delete, and Insert operations. The functional specifications worked by combining node invariant specifications and list manipulation predicates.
to specify the physical and logical data structure state respectively.

We establish that the different specification components can be generated individually from the parameters of each design element and can be combined as necessary to specify the overall data structure. The modular nature of the node specifications suggests that synthesis of a full data structure implementation may be split up into the smaller tasks of synthesizing functions for individual types of nodes.

To the best of our knowledge, Cozy is the latest and perhaps the only general data structure synthesis tool available currently. It works by composing together simpler data structures like arrays and hash maps to generate more complex data structure designs. This approach is fundamentally different from ours as the data structure specifications we generate are based on the Data Calculator design primitives which “represent fundamental design choices” and thus boast greater expressivity of data structure designs.

Aside from the broad goal of data structure synthesis, the generated data structure specifications may be employed to formally verify implementations of Data Calculator designs. Since they are interpretable logical formulas which show how different parameter settings translate to implementation choices, the specifications will likely be helpful for engineers who are seeking to manually implement Data Calculator designs too.
References


