SEQUENCE TREES: LOGARITHMIC SLICING AND CONCATENATION OF SEQUENCES

RODNEY M. BATES

Department of Computer Science, Wichita State University
Wichita, KS 67260-0083, USA
Tel: 316-978-3922
Fax: 316-978-3984

ABSTRACT. We describe a concrete data structure, called a sequence-tree, that represents sequences of arbitrary elements, along with associated algorithms that allow single element access and assignment, subsequence extraction (slicing), and concatenation to be done in logarithmic time relative to sequence length. These operations are functional, in the sense that they leave their operand sequences unchanged. For a single sequence, space is linear in the sequence length. Where a set of multiple sequences have been computed by these algorithms, space may be sublinear, because of node sharing. Sequence-trees use immutable, shared, dynamically allocated nodes and thus may require garbage collection, if some of the sequences in a set are abandoned. However, the interconnection of nodes is non-cyclic, so explicity programmed collection using reference counting is reasonable, should a general-purpose garbage collector be unavailable. Other sequence representations admit only to linear-time algorithms for one or more of the aforementioned operations. Thus sequence-trees give improved performance in applications where all the operations are needed.

1. INTRODUCTION

A *sequence-tree* is a concrete data structure which represents an abstract *sequence.* Sequence-trees allow single element access and assignment, subsequence extraction (slicing), and concatenation to be done in logarithmic time. Furthermore, all these operations leave their operand sequences unchanged. For a single sequence, space is linear. For multiple sequences, space can be sublinear.

Sequence-trees use immutable nodes, which can be shared among sequence values. Thus they may require garbage collection, either as part of the underlying programming system, or explicitly programmed as part of the sequence-tree implementation. Sequence-trees are non-cyclically linked, so simple reference counting will work.

Sequence-trees are useful in applications where both subscripting and either slicing or concatenation operations are performed on sequences. Their greatest value is in situations where the maximum sequence length is large and asymptotic behavior dominates. They were developed in response to a specific need, during implementation of a semantic programming editor that internally maintains a variation of an abstract syntax tree, while providing a text-editor style user interface.

Other concrete representations of sequences are generally variations on either arrays or linked lists. Discussions of these can be found in standard data structures and algorithms texts, e.g. [1], [2], [4], [8], [9]. Sequence-trees maintain logarithmic complexity in all of the sequence operations discussed while preserving operand values. The many variations on array and linked representations are all linear in one or more of these operations. Most do not preserve operand values.

In their concrete representation, sequence-trees resemble B-trees [2], [4], [9]; 2-3 trees [3], [4], [9]; and (a,b) trees [10]. All these are height-balanced trees whose nodes have varying but bounded out-degree. Huddleston and Mehlhorn [10] give a set of invariants for (a,b) trees which are equivalent to those stated here for sequence-trees.

However, sequence-trees differ from these others in the abstraction they implement. B-trees etc. implement *dictionaries*, also called *sorted sets*. That is, they map a possibly sparse, finite set of search keys (chosen from some totally ordered set) to elements, where the association between a key and its element is not altered by changes elsewhere in the mapping. The search trees for these mappings are always sorted on the key values. Thus, concatenation is not a meaningful operation, since it would not in general preserve the sorting. Alterations to dictionaries are limited to single element insertions and deletions.

In contrast, sequence-trees implement *sequences*, whose elements are located by compactly numbered subscripts. These can be viewed as mappings from a finite range of integers to sequence elements, but the mapping is always implied by the order of the elements. Thus, any insertions or deletions of sequence elements will shift the mapping from subscripts to elements, for elements located to the right of the leftmost change. The element values in a sequence are not sorted and need not have an ordering defined. Their order is instead determined by the series of operations creating the sequence.

There are many other tree structures that allow logarithmic search, e.g. AVL trees [2], [8], [9]; weight balanced trees [12], red-black trees [7], and weak B-trees [10]. All these represent dictionaries rather than sequences, and all handle insertions and deletions of single elements only. Aho, Hopcroft and Ullman [1] mention an operation CONCATENATE on

2-3 trees. This works on dictionaries and has a different meaning from that used here, i.e. CONCATENATE requires that all keys in the left operand be less than all those in the right operand.

Several authors have given tree structures which preserve operands during modifications, by copying nodes on a path, e.g. [11] and [13]. Most of these are again for dictionaries rather than sequences. Dobkin and Munro [6] give a representation of sequences which also preserves operands of sequence-modifying operations, but it requires that all modifications precede all accesses. Sarnak and Tarjan [14] briefly mention the application of path copying to implement sequences, but do not suggest slicing or concatenation as operations.

Dagenais [5] offers a Modula-3 module implementing sequences which uses the sequence-tree data structure. However, it does not preserve operands, thus slicing and concatenation, which are not provided, could not be done with logarithmic time. Instead, only single element insertion and deletion are provided.

The rest of this paper is organized as follows. In section 2, we define the sequence abstraction that sequence-trees will be used to represent. In section 3, we define the concrete sequence-tree data structure, along with the invariants it must satisfy, and give the mapping from sequence-trees to the abstract sequences they represent. Section 4 defines some support algorithms on sequence-trees that will be needed in the later sections. Sections 5 and 6 describe the single-element access and assignment algorithms. Section 7 describes the concatenation algorithm and section 8 describes the slicing algorithm. Finally, section 9 discusses some pragmatic aspects of the data structure and algorithms and briefly relates some composites of the basic algorithms presented.

2. The Sequence Abstraction

Let Σ be a set. We will call it the *alphabet*, and sequences will have members of Σ as elements. We define Σ^* to denote the set of finite sequences over Σ. Let $s = s_0, s_1, \ldots, s_{q-1} \in \Sigma^*$. We say that s is a sequence of length q, which we write as $|s| = q$. We use ϵ to denote the empty sequence, i.e. the sequence with no elements. We also define the sets of sequences over Σ with finite minimum and maximum length:

$$\Sigma^{p,q} = \{s \in \Sigma^* \mid p \leq |s| \leq q\}$$

Definition 2.1. assign

The *element assignment function* is defined as:

$$\text{assign} : \Sigma^* \times \aleph \times \Sigma \rightarrow \Sigma^*$$

$$\begin{aligned}
\text{assign}(s,i,x) &= \begin{cases} s_0, s_1, \ldots, s_{i-1}, x, s_{i+1}, \ldots, s_{q-1} & i < q \\ s & \text{otherwise} \end{cases} \\
\text{where s} &= s_0, s_1, \ldots, s_{q-1} \text{ and } |s| = q
\end{aligned}$$

Definition 2.2. cat

The *concatenation function* is defined as:

$$\text{cat} : \Sigma^* \times \Sigma^* \to \Sigma^*$$

$$\begin{aligned}
\text{cat}(r,s) &= r_0, r_1, \ldots, r_{p-1}, s_0, s_1, \ldots, s_{q-1} \\
\text{where } r &= r_0, r_1, \ldots, r_{p-1} \text{ and } |r| = p \\
\text{and } s &= s_0, s_1, \ldots, s_{q-1} \text{ and } |s| = q
\end{aligned}$$

Definition 2.3. slice

The *slice function* is defined as:

$$\text{slice} : \Sigma^* \times \aleph \times \aleph \to \Sigma^*$$

$$\begin{aligned}
\text{slice}(s,i,j) &= \begin{cases} s_i, s_{i+1}, \ldots, s_{i+j} & i + j < q \\ s_i, s_{i+1}, \ldots, s_{q-1} & i + j \leq q, i < q \\ \epsilon & i \geq q \end{cases} \\
\text{where } s &= s_0, s_1, \ldots, s_{q-1} \text{ and } |s| = q
\end{aligned}$$

cat is associative, and we will sometimes write it as the infix operator $\oplus$, e.g. $s_0 \oplus s_1 \oplus s_2$.

3. Representation

In this section, we define the sequence-tree representation of sequences, the invariants it must satisfy, and the representation function that maps a sequence-tree to the abstract sequence it represents.

Intuitively, a sequence-tree is a possibly nil pointer to a *node.* A node is either a *leaf node* or a *nonleaf node.* A leaf node is just a small array whose elements are members of the alphabet Σ and which represents itself as a sequence. It has at least two and at most d elements. d is a small constant, not less than 3, which is the maximum degree of any node. We have used $d = 8$ in actual implementations. If the sequence elements are large or variable in size, they can be actually represented in leaf nodes as pointers to sequence elements.

A *nonleaf node* contains a positive integer field named Height and an array whose element count also lies between 2 and d. Each of its elements contains a positive integer field named CumChildCt and a never-nil pointer, named ChildRef, to a subtree, which is a node. A nonleaf node represents the concatenation of the sequences represented by its children.

A sequence-tree is height-balanced, meaning the paths to each leaf of a given tree have the same length, called the *height* of the tree. The height of a leaf node is implicitly one, and the height of a nonleaf node is stored (redundantly) in the Height field.

The *length* of a leaf node is its element count and the length of a nonleaf node is the sum of the lengths of its subtrees. The value of CumChildCt of an element of a nonleaf node (redundantly) contains the sum of the lengths of the subtree rooted at this element and of all elements to its left in the same nonleaf node.

As special cases, the nil pointer represents the empty sequence and a pointer to a single sequence element represents a singleton sequence. Alternatively, this can be viewed as a leaf node which has only one element, violating the usual minimum degree of 2 for nodes. Some of the algorithms will treat a singleton identically with a leaf node. These special cases can only occur as entire sequence-trees, not as proper subtrees accessible from a nonleaf node.

3.1. Concrete Data structure.

We recursively define the following sets:

$$\begin{aligned} L &= \Sigma^{2,d} \\ E &= \aleph \times M \\ A &= E^{2,d} \\ N &= \aleph \times A \\ M &= L \cup N \\ T &= M \cup \Sigma \cup \{\Lambda\} \end{aligned}$$

Λ denotes a nil pointer and $d \geq 3$ is the maximum node degree described above. Here, L is the set of leaf nodes, E is the set of nonleaf elements, A is the set of arrays of nonleaf elements, whose length lies between 2 and d, N is the set of nonleaf nodes, M is the set of nodes, and T is the set of sequence-trees. For consistency with the subscripting of sequence elements, we also use zero-origin subscripts to distinguish the elements of the two cartesian products above. Thus, e.g., if $e \in E$, then $e_0 \in \aleph$. For readability in larger expressions, we will sometimes use the field names introduced above to denote the elements of the cartesian products:

$$\begin{aligned} \text{CumChildCt}(e) &\equiv e_0 \\ \text{ChildRef}(e) &\equiv e_1 \\ \text{Height}(n) &\equiv n_0 \\ \text{Elems}(n) &\equiv n_1 \end{aligned}$$

where $e \in E$ and $n \in N$. As usual with recursive data structures in computer programs, ChildRef, as the name suggests, is actually represented by a pointer to a member of M.

Definition 3.1. rep

We define the *representation function* by

$$\text{rep} : T \to \Sigma^*$$

$$\text{rep}(t) = \begin{cases} \epsilon & t = \Lambda \\ t & t \in \Sigma \\ t & t \in L \\ \sigma & t = (k, (n_0, n_1, \ldots, n_{p-1})) \in N \end{cases}$$

where $\sigma = \text{rep}(n_{0,1}) \oplus \text{rep}(n_{1,1}) \oplus \cdots \oplus \text{rep}(n_{p-1,1})$ and $p = |\text{Elems}(t)|$

We will refer to the four cases of the definition of rep as the *empty case*, *singleton case*, *leaf case*, and *nonleaf case*, respectively.

Definition 3.2. Count field rule

The CumChildCt field of a nonleaf element a_i of array $a \in A$ satisfies the *count field rule* if:

$$\text{CumChildCt}(a_i) = \sum_{j=0}^{i} |\text{rep}(\text{ChildRef}(a_j))|$$

A member of T satisfies the count field rule if every CumChildCt field it contains satisfies the count field rule.

Informally, the count field rule says that the CumChildCt field of a nonleaf node element contains the sum of the lengths of the sequences represented by the subtree rooted at this element and all elements to its left in this nonleaf node. This field is redundant, but is required by the algorithms we describe. An obvious algorithm, of $O(1)$ complexity, to compute $|\text{rep}(t)|$, $t \in T$, where t satisfies the count field rule, follows from the definition of the representation function, utilizing, in the nonleaf case, the CumChildCt field of the last element of the nonleaf node and the fact that the length of a concatenation is the sum of the lengths of the constituents. We will denote this algorithm Length : $T \to \aleph$.

Definition 3.3. leftCount

The *left count* function is defined as:

$$\text{leftCount} : A \times \aleph \to \aleph$$

$$\text{leftCount}(a, i) = \sum_{j=0}^{i-1} |\text{rep}(\text{ChildRef}(a_j))|$$

An obvious algorithm, which we will call LeftCount $(A \times \aleph) : \aleph$, of complexity $O(1)$, exists to compute leftCount. It uses CumChildCt(a_{i-1}), with a special case for $i = 0$.

Definition 3.4. height

The *height function* on sequence-trees is defined as:

$$\text{height} : T \to \aleph$$

$$\text{height}(t) = \begin{cases} 0 & t = \Lambda \\ 1 & t \in L \cup \Sigma \\ 1 + \text{height}(\text{ChildRef}(\text{Elems}(t)_0) & t \in N \end{cases}$$

Definition 3.5. Height balance condition

A nonleaf node $n \in N$ satisfies the *height balance condition* if:

$$\text{height}(\text{ChildRef}(a_j)) = \text{height}(\text{ChildRef}(a_0))$$
$$\text{where } a = \text{Elems}(n),\ \forall j | 0 \leq j < |a| - 1$$

A member of T satisfies the height balance condition if every nonleaf node it contains satisfies the height balance condition.

Definition 3.6. Height field rule

The Height field of a nonleaf node n satisfies the *height field rule* if :

$$\text{Height}(n) = \text{height}(n)$$

Informally, this rule says that the Height field of a nonleaf node contains the height of the subtree rooted at this node. This field is also redundant, but is also required by the algorithms. An obvious algorithm, which we will call Height(T) : $\aleph$, of $O(1)$ complexity, to compute height of a member of T which satisfies the height balance and height field rules, follows directly from the definition of height, using the Height field in the nonleaf node case.

Definition 3.7. Well formedness of sequence-trees

We say a member of T is *well formed* if it satisfies the count field rule, the height balance condition, and the height field rule. We call a well formed member of T a *sequence-tree.*

Because leaf nodes have height=1 and nonleaf nodes have height>1, it follows from the height balance condition that, in a sequence-tree, any nonleaf node contains, in its ChildRef fields, either all leaf nodes or all nonleaf nodes, but not a mixture.

Theorem 3.8. *The height bound theorem*

Let $t \in T$ be well formed, $s = \text{rep}(t)$, $|s| > 1$. Then $\text{height}(t) \leq \lfloor \log_2 |s| \rfloor$

Proof. The empty and singleton cases are excluded by the premise $|s| > 1$. For the leaf case, $|s| \geq 2$ implies $\log_2 |s| \geq 1$ and $\lfloor \log_2 |s| \rfloor \geq 1$, while $\text{height}(t) = 1$. These together satisfy the conclusion of the theorem. For the nonleaf case, we prove the theorem by induction on tree height. The leaf case above is also the basis case for the induction. For the induction step, assume the theorem holds for every sequence-tree u with height(u) $< h$. Thus $2^{h-1} \leq |\text{rep}(u)|$. Now h has at least two subtrees u_0 and u_1, and $|\text{rep}(u_0)| + |\text{rep}(u_1)| \leq |s|$. Then $2^{h-1} + 2^{h-1} = 2^h \leq |s|$, and $h \leq \log_2 |s| \leq \lfloor \log_2 |s| \rfloor$, since h is integral. □

Definition 3.9. sch

The *search function*, defined on nonleaf nodes, is given by

$$\text{sch} : N \times \aleph \rightarrow \aleph$$

$$\text{sch}(n, i) = \text{the smallest } j \text{ such that i} < \text{CumChildCt}(\text{Elems}(n)_j)$$

An algorithm Sch($N \times \aleph$) : $\aleph$ to compute sch can be constructed, using classical binary search of the array Elems(n). Although nontrivial, this is a well-known algorithm that we will not repeat here. Its complexity is $O(\lceil \log_2(p) \rceil)$, where p is the number of elements of the array. This is bounded by $O(\lceil \log_2(d) \rceil)$, which is constant, thus Sch is $O(1)$. Simple linear search would also have constant time complexity, but the binary search has a better constant factor.

3.2. **Comments on the Representation.** Fig. 3.1 shows a diagrammatic notation. Part (a) is a leaf node, showing an array of four elements, together with their subscripts written above. Part (b) shows a nonleaf node, with its Height field at the left, separated by a bold line from the Elems field. This is an array of degree three, with its subscripts shown above. The CumChildCt and ChildRef fields of each nonleaf element are shown, separated by a dashed line.

Fig. 3.2 shows several example sequence-trees, using the node notation of Fig. 3.1, together with the sequence each represents. Examples (b) and (c) are different sequence-tree representations of the same sequence.

The nodes are linked together in a tree using only child pointers in the ChildRef fields. There are no sibling or parent pointers. Thus no node contains any inherited information. Furthermore, nodes are immutable, that is, once created, a node is never changed. These two properties mean that a node, and therefore a sequence-subtree, can be shared among many parents, each of which belongs to a different sequence-tree.

If each node has exactly two elements, the sequence-tree resembles a binary tree. Since nodes may have more than two children, the height can be significantly less. In fact, if the degree of every node is d, then the sequence-tree height is $O(\log_d(|s|))$. In the worst case, where each node has only two elements, there are $O(\log_2(|s|) - 1$ nonleaf levels containing a total of $|s| - 2$ elements, so total space occupied is $O(|s|)$.

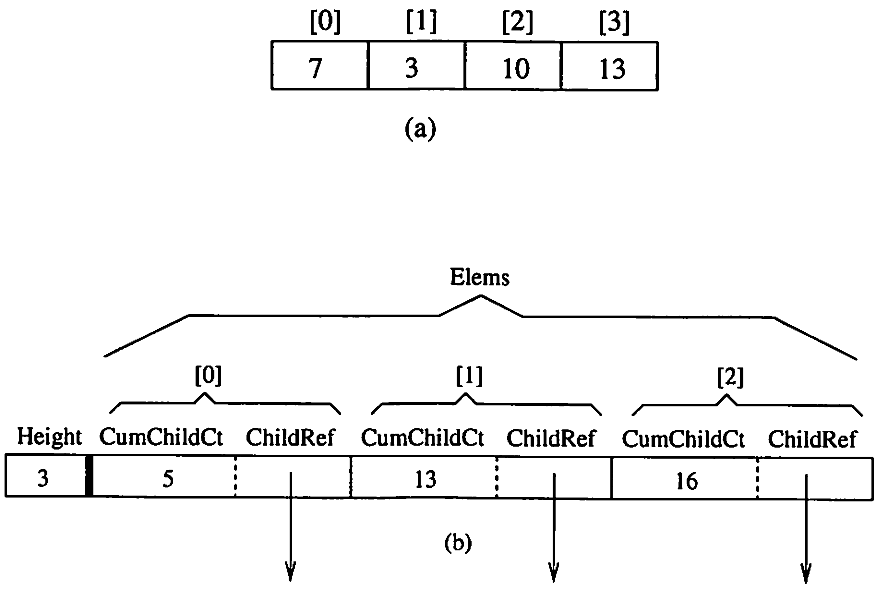

FIGURE 3.1. Sequence-tree node notation

4. SUPPORT ALGORITHMS

Besides those already mentioned in the definitions, we will need the following support functions in the algorithms to be specified.

Algorithm 4.1. *MakeNonleaf*

We assume the existence of algorithm MakeNonleaf(Elems : $M^{2,d}$) : N which takes a sequence of members of M (i.e. pointers to either leaf or nonleaf nodes) and constructs a nonleaf node from them, properly computing the Height and CumChildCt fields. This construction is mechanical and bounded in complexity by d, and thus $O(1)$. We will not show its algorithm, but note that, in order to be able to construct a node of a sequence-tree, it will require as a precondition, that |Elems| lie in the range $2 \cdots d$ and that the elements of Elems all have the same height.

Algorithm 4.2. *MakeNonleafPair*

The algorithm MakeNonleafPair(m : $M^{2,2d}$) : $N \times (N \cup \Lambda)$ accepts a string of leaf or nonleaf elements and constructs either one or two new nonleaf nodes, whose children are the supplied elements. It has preconditions that all such elements have the same height and that the length of the string is at most $2d$. If only one nonleaf node is required to hold the elements, it returns Λ as the second member of the pair it computes.

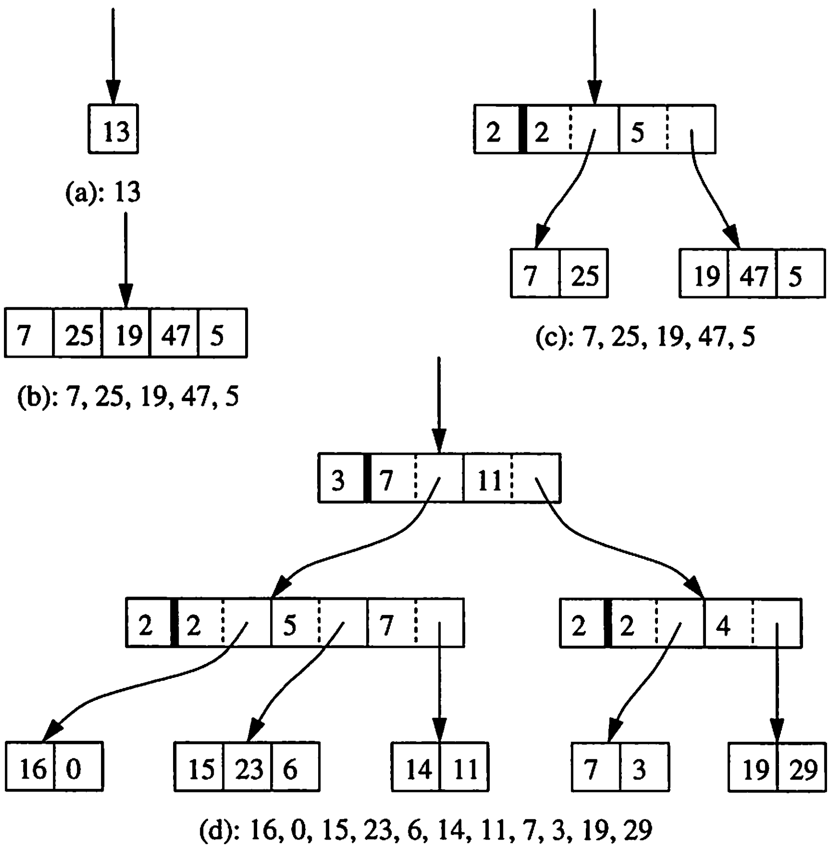

FIGURE 3.2. Sequence-tree examples

```
FUNCTION MakeNonleafPair
  ( m : M ) : ( N × ( N ∪ Λ ) )
= LET ct := LEN m
  IN IF ct <= d
     THEN ( MakeNonleaf ( m ) , Λ )
     ELSE LET j := ct DIV 2
       IN  ( MakeNonleaf ( m [ 0 ] ⊕ ··· ⊕ m [ j - 1 ] )
           , MakeNonleaf ( m [ j ] ⊕ ··· ⊕ m [ ct - 1 ] )
           )
```

In the algorithms, `LEN` is a prefix operator that returns the length of an array, `DIV` is integer division with truncation, and $\oplus$ is concatenation of arrays and/or single elements into a longer array.

Theorem 4.3. *Correctness and complexity of MakeNonleafPair*

Let $m = m_0, \dots, m_{c-1} \in M^{2,2d}$, $|m| = c$, and $\exists h$ such that, $\forall m_p$, m_p is a sequence-tree, and $\text{height}(m_p) = h$, for $0 \leq p < c$. Further, let $(l, r) =$ MakeNonleafPair(m). Then

(1) l and r are sequence-trees
(2) $\text{height}(l) = h + 1$
(3) either $r = \Lambda$ or $\text{height}(r) = h + 1$
(4) $\text{rep}(l) \oplus \text{rep}(r) = \text{rep}(m_0) \oplus \cdots \oplus \text{rep}(m_{c-1})$
(5) MakeNonleafPair terminates in constant time.

Proof. The precondition that all elements of m have the same height ensures height balance of the one or two constructed new nodes. The calls on MakeNonleaf ensure that these satisfy the height field and count field rules. These, plus well formedness of the elements of m ensure well-formedness of the results. Height of the results follows directly from the construction. The condition on the representations follows directly from the node construction also.

We must show that the precondition of MakeNonleaf is satisfied. In the **then** case of the `if` expression, the lower bound follows from the precondition to MakeNonleafPair, and the upper limit follows directly from the condition of the `if` statement. In the `else` case, $\text{ct} > d \geq 3$, thus $\text{ct} \geq 4$, and $j = \text{ct} \operatorname{div} 2 \geq 2$. Also, $2j = j + j \leq \text{ct}$ from computation of j, so $j + 2 \leq \text{ct}$, and the two constructed leaf nodes have element counts of at least two. By precondition to MakeNonleafPair, $\text{ct} \leq 2d$, so $j \leq d$ and $\text{ct} - j \leq d$, and thus the two new nodes satisfy the upper limit of node size.

The concatenations in MakeNonleafPair are bounded by the constant d, and MakeNonleaf has constant complexity, so MakeNonleafPair does also. $\square$

Algorithm 4.4. *MakeLeafPair*

The algorithm MakeLeafPair$(e : \Sigma^{2,2d}) : (L \times (L \cup \Lambda))$ is similar to, but simpler than MakeNonleafPair. It accepts a sequence of abstract sequence elements with the same length bounds as MakeNonleafPair and returns a pair of leaf nodes. Its code is just like MakeNonleafPair except it does not call MakeNonleafNode or any counterpart to it. Its correctness theorem and proof are also just like MakeNonleafPair.

5. Subscripting

The algorithm Ss returns the ith element of the sequence represented by a sequence-tree. Informally, Ss proceeds top-down, using a subscript which is always relative to the current sequence-subtree. The empty and singleton sequence cases are straightforward. If the sequence-subtree is a leaf, the sequence subscript is the subscript to the node and leads to the desired sequence element in constant time. If the sequence-subtree is a nonleaf, Ss chooses the child which must contain the sought sequence element and descends into that subtree. It uses Sch to make this choice. The sought element will, in the subtree, have a different subscript, reduced by the number of sequence elements in subtrees to the left, which LeftCount will provide.

Algorithm 5.1. *Ss*

```
FUNCTION Ss ( t : T ; i : ℵ ) : Σ ∪ ϵ
= IF i = Length ( t ) THEN ϵ
  ELSIF t ∈ Σ THEN t
  ELSIF t ∈ L THEN t [ i ]
  ELSE LET
      a := Elems ( t )
      j := Sch ( a , i )
    IN Ss ( ChildRef ( a [ j ] )
          , i - LeftCount ( a , j )
          )
```

Theorem 5.2. *Correctness and complexity of Ss*

Ss(t, i) computes $(\text{rep}(t))_i$ in time bounded by $O(\log_2(|\text{rep}(t)|)$

Proof. Only the nonleaf case is nontrivial. Consider the point where the recursive call on Ss is about to be evaluated. Using the value names in the algorithm, define the following:

$$\begin{aligned}
c_j &= \text{leftCount}(a, j) \\
c_{j+1} &= \text{leftCount}(a, j+1) \\
c &= c_{j+1} - c_j \\
r &= \text{rep}(\text{ChildRef}(a_j)) = r_0, r_1, \ldots, r_{c-1} \\
s &= \text{rep}(t)
\end{aligned}$$

$c_j \le i < c_{j+1}$, from the definitions of leftCount and sch. From the definition of rep(t), we have:

$$s = s_0, \oplus \ldots \oplus, s_{c_j-1} \oplus s_{c_j}, \oplus \ldots, \oplus s_{c_{j+1}-1} \oplus s_{c_{j+1}}, \oplus \ldots, \oplus s_{|s|-1}$$

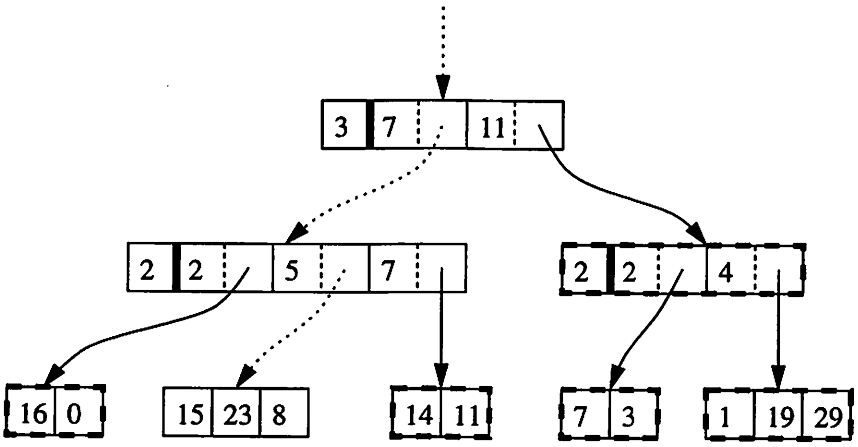

FIGURE 6.1. After single element assignment

By correspondence of elements, $r_k = s_{c_j+k}$, for $0 \leq k < c$. Choose $k = i - c_j$. Then

$$r_{i-c_j} = s_i$$

The left side of this equality is what the recursive call on Ss computes and the right side is what we desire.

When the algorithm is applied to a nonleaf node of height h, the recursive call is applied to a node of height $h - 1$. Thus the algorithm terminates and has complexity $O(h) \leq \log_2 |\text{rep}(t)|$, by the height bound theorem, since the other functions evaluated, Sch and LeftCount have constant time complexity. □

6. Single Element Assignment

Element assignment begins like single element access, proceeding top-down through the sequence-tree to the correct element of each node. However, the located element cannot be altered in place, because this would violate node immutability. Instead, the assignment operation allocates a copy of the old leaf node and alters the ith element of the copy. It then returns the pointer to the new node to the level above. Each nonleaf level does essentially the same thing, except it replaces only the ChildRef field of the selected element of its copied node with the pointer it receives from below. The Height and CumChildCt fields do not need to be recomputed in the copied node.

The result is a new sequence-tree in which all nodes on the path from the root to the leaf node containing the ith sequence element have been replaced, while all nodes off this path are shared with the original sequence-tree. The complexity is proportional to the sequence-tree height, times the maximum node element count d. This is bounded by $O(\log_2(|\text{rep}(t)|))$.

Algorithm 6.1. *Assign*

```
FUNCTION Assign ( t : T ; i : $\aleph$ , val : $\Sigma$ ) : T
= IF i $\geq$ Length ( t )
  THEN t
  ELSIF Height ( t ) = 1
  THEN (* leaf/singleton case *)
      t [ 0 ] $\oplus \cdots \oplus$ t [ i - 1 ]
      $\oplus$ val
      $\oplus$ t [ i + 1 ] $\oplus \cdots \oplus$ t [ LEN t - 1 ]
  ELSE (* nonleaf case *)
    LET
      a := Elems ( t )
    , j := Sch ( a , i )
    , k := i - LeftCount ( a , j )
    , n := Assign ( ChildRef ( a [ j ] ) , k , val )
    , new
      := ChildRef ( a [ 0 ] )
         $\oplus \cdots \oplus$ ChildRef ( a [ j - 1 ] )
         $\oplus$ n
         $\oplus$ ChildRef ( a [ j + 1 ] )
         $\oplus \cdots \oplus$ ChildRef ( a [ LEN a - 1 ]  )
    IN MakeNonleaf ( new )
```

Theorem 6.2. *Correctness and complexity of Assign*

Let $s = \text{Assign}(t, i, v)$. Then $\text{rep}(s) = \text{assign}(\text{rep}(t), i, v)$ and Assign completes in time bounded by $O(\log_2(|\text{rep}(t)|)$

Proof. The proof directly mirrors that of Ss, with, at both the leaf and nonleaf levels, construction of a new sequence with one element replaced, instead of selecting the desired element. □

6.1. **An assignment example.** Fig. 6.1 shows the result of assigning the value 8 to the sequence element whose subscript is 4, in the sequence-tree of Fig. 3.2(d). The nodes boxed in heavy dashes have identical contents and are the same nodes as before the assignment. The other nodes are new. The two new nonleaf nodes look the same as before the assignment, but each differs in having a different value in one of its ChildRef fields. To illustrate this, pointer values which have changed since the assignment are shown as dotted arrows.

7. Concatenation

At least for sequence-trees of equal height, there is a naive algorithm for concatenation that just constructs a new nonleaf node whose two children are the operand sequence-trees. In order to handle trees of unequal height and also, to attempt to keep nodes more nearly full, we give a more sophisticated algorithm.

Concatenation of the sequences represented by two sequence-trees is performed by constructing a *seam* along the right edge of the left operand sequence-tree and the left edge of the right operand sequence-tree. The seam is constructed bottom up, matching and possibly joining pairs of nodes of the same height from the two sides of the seam. At each level in the seam, one or two new nodes may be allocated. All other nodes in the two operand sequence-trees are shared with the result sequence-tree.

At the leaf level, if the total number of elements from the two seam nodes of a pair is $\leq d$, these elements are all repacked into a single, new leaf node and the pointer to this is returned to the level above, to be incorporated into the seam at that level. The two old leaf nodes are left alone, since they belong to the operand sequence-trees. Even if both the operand leaf nodes are singleton sequences, there will be enough elements to satisfy the two-element minimum.

If the total number of elements in the two leaf nodes is $> d$, the pointers to the two old leaf nodes are both returned, again to be incorporated into the seam at the level above. In this case, each of these two nodes will be shared between the value of the concatenated result and one of the operands.

At nonleaf levels, the procedure must be generalized somewhat. As before, a pair of nodes from the edges of the operand trees is being matched and/or joined. If the two nodes at the level below are being reused intact, then the two nodes at this level are treated exactly as at the leaf level, i.e. they are repacked into one new node if possible or reused intact otherwise.

If any repacking was done at the level below, then the sequence of subtrees needed for the seam at this level consists of all but the rightmost child of the left node, the one or two new sequence-subtree(s) returned from below, and all but the leftmost child of the right node. These are collected into either one or two new nodes, as required to hold them. The total number of children will be at least 3 and at most $2d$, so one or two new nodes will always suffice and the bounds on node degree can always be satisfied.

If one of the operand sequence-trees of a concatenation is lower than the other, there can be nonleaf levels where there is no existing node on one side of the seam. At all such levels, the concatenation algorithm behaves as though the low operand sequence-tree were extended with a fictitious nonleaf node of degree one, whose single child points to the fictitious or real sequence-tree node below. Furthermore, at such levels, repacking is always performed, so that the fictitious node, which would otherwise violate the two-element minimum, will never actually be constructed. Even in this case, there will be at least 2 elements to be repacked at this level.

Although the leaf level can return only one new sequence-subtree, a nonleaf level can return one or two, so the seam construction at a nonleaf level assumes for generality that it can receive two new sequence-subtrees from below. At height ≥ 3, this is possible.

Above the topmost level of the operand sequence-trees, either one or two children will be returned. If there is only one, this is the result of the concatenation. Otherwise, a new, two-element nonleaf node must be constructed, with these as its children, which will be the result.

The work of the concatenation algorithm proceeds bottom-up. However, since sequence-trees have only child pointers, all algorithms must begin at the top of a sequence-tree, descending recursively to the bottom, and doing the node copying and repacking on the way back up. The Height fields in sequence-tree nodes are needed so that the algorithm can properly locate the pairs of nodes of equal height on either side of the seam. This height synchronization must be established during the downward phase.

The complete concatenation algorithm has a top level function Cat and a recursive function CatRecurse.

Algorithm 7.1. *Concatenation*

```
FUNCTION Cat ( left , right : T ) : T
= IF left = Λ THEN right
  ELSIF right = Λ THEN left
  ELSE LET ( newLeft , newRight )
           := CatRecurse ( left , right )
  IN IF newRight = Λ
    THEN (* case1 *) newLeft
    ELSE (* case2 *)
      MakeNonleaf ( newLeft ⊕ newRight )
```

Algorithm 7.2. *CatRecurse*

```
FUNCTION CatRecurse ( left , right : M ∪ Σ ) : ( T × T )
= LET leftHeight := Height ( left )
  , rightHeight := Height ( right )
  , h := MAX ( leftHeight , rightHeight )
  , lct := Length ( left )
  , rct := Length ( right )
  IN IF h = 1
    THEN (* the leaf/singleton case *)
    LET ct := lct + rct
    IN IF ct $\leq$ d OR lct < 2 OR rct < 2
      THEN (* leaf1 *) MakeLeafPair( left $\oplus$ right )
      ELSE (* leaf2 *) ( left , right )
  ELSE (* h > 1, the nonleaf case *)
    LET ( leftContext , leftDesc )
        := IF leftHeight < h
           THEN ( $\epsilon$ , left )
           ELSE ( ChildRef ( Elems ( left ) [ 0 ] )
                  $\oplus \cdots \oplus$
                  ChildRef ( Elems( left ) [ lct - 2 ] ) )
                , ChildRef ( Elems ( left  ) [ lct - 1 ]
                )
    , ( rightContext, rightDesc )
        := IF rightHeight < h
           THEN ( $\epsilon$ , right )
           ELSE ( ChildRef ( Elems ( right ) [ 1 ] )
                  $\oplus \cdots \oplus$
                  ChildRef ( Elems ( right ) [ rct - 1 ] ) )
                , ChildRef ( Elems ( right ) [ 0 ]
                )
    , ( newLeft, newRight )
        := CatRecurse ( leftDesc , rightDesc )
    , new
        := IF newRight = $\Lambda$
           THEN leftContext $\oplus$ newLeft $\oplus$ rightContext
           ELSE leftContext $\oplus$ newLeft $\oplus$ newRight
                $\oplus$ rightContext
    IN IF LEN new $\leq$ d
          OR leftContext = $\epsilon$
          OR rightContext = $\epsilon$
          OR newLeft $\neq$ leftDesc
          OR newRight $\neq$ rightDesc
      THEN (* nonleaf1 *) MakeNonleafPair ( new )
      ELSE (* nonleaf2 *) ( left , right )
```

We will use the labels enclosed in comments in the algorithm to denote various cases. Most of the proof of this algorithm lies in the following lemma, which expresses preconditions and postconditions for calls on CatRecurse.

Lemma 7.3. *Correctness of CatRecurse*

Let $l_0 \in M \cup \Sigma$, and $r_0 \in M \cup \Sigma$ be well-formed. Further, let (l,r) = CatRecurse (l_0, r_0). Then

(1) $l \in M$ is well-formed.
(2) $r \in M \cup \Lambda$ is well-formed.
(3) $\text{rep}(l) \oplus \text{rep}(r) = \text{rep}(l_0) \oplus \text{rep}(r_0)$.
(4) $\text{Height}(l) = \max(\text{Height}(l_0), \text{Height}(r_0))$.
(5) either $r = \Lambda$ or $\text{Height}(r) = \text{Height}(l)$.

Proof. We will prove the lemma by induction on tree height.

Basis case, i.e. height = 1. This is the leaf case, $l_0 \in L \cup \Sigma$ and $r_0 \in L \cup \Sigma$. From the definitions of L and Σ, ct ≥ 2.

In case leaf1, if ct $> d$, then either lct = 1 or rct = 1, so ct $\leq d+1 < d+2$, thus ct $\leq 2d$, and the preconditions of MakeLeafPair are satisfied. Leaf nodes trivially satisfy well-formedness and are all of the same height. The postconditions of MakeLeafPair ensure the remaining postconditions of the lemma.

In case leaf2, the result pair is exactly the argument pair, and the postconditions follow directly.

Induction step: Assume the lemma holds for sequence-trees of height $h - 1$, $h > 1$. For height h, this is a nonleaf case, i.e. $\text{Height}(l_0) > 1$ or $\text{Height}(r_0) > 1$. Prior to the recursive call, we have

$$\begin{aligned} \text{rep}(l_0) &= \text{rep(leftContext)} \oplus \text{rep(leftDesc)} \\ \text{rep}(r_0) &= \text{rep(rightDesc)} \oplus \text{rep(rightContext)} \end{aligned}$$

by either of the alternate constructions of leftContext and rightContext. leftDesc and rightDesc satisfy the precondition for the recursive call, either from the precondition of this invocation of CatRecurse or from the definition of N. Appealing to the inductive assumption, after new has been computed,

$$\text{rep(new)} = \text{rep}(l_0) \oplus \text{rep}(r_0)$$

In case nonleaf1, by inductive assumption, $|\text{newLeft}| \geq 2$, and thus $|\text{new}| \geq 2$. Also, $|\text{new}| \leq 2 + 2(d-1) = 2d$, and $|\text{new}|$ satisfies the length preconditions of MakeNonleafPair. Elements in new that came from leftContext or rightContext are well-formed by assumption of the lemma, and those from newLeft and newRight are well-formed by the inductive assumption for the recursive call. Similarly, any elements in new that came from leftContext or rightContext have height h by their construction, and those from newLeft and newRight have height h by the inductive assumption for the recursive call and the omission of newRight from new when it equals Λ. This satisfies all the preconditions of MakeNonleafPair, whose postconditions directly imply the desired conclusions of the lemma.

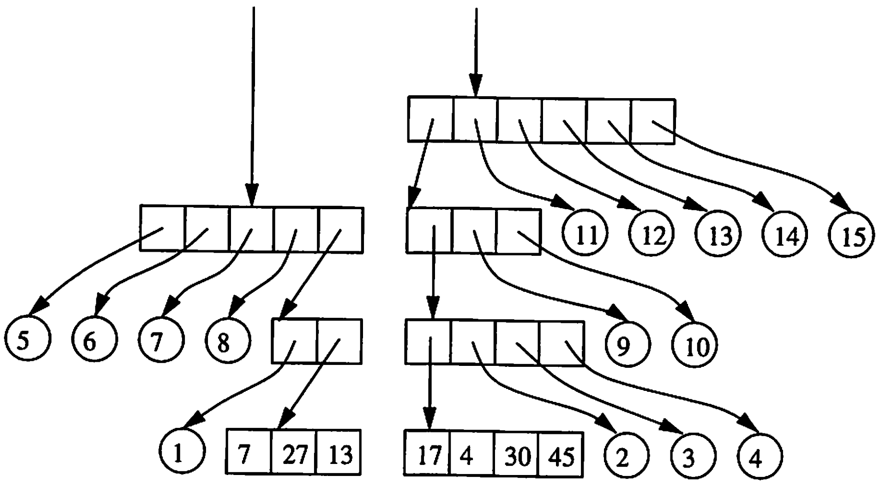

FIGURE 7.1. Before concatenation

In case nonleaf2, the results of the algorithm equal its arguments. The 4th conclusion of the lemma, on the height of the results, follows from leftContext $\neq \epsilon$ and rightContext $\neq \epsilon$, which are necessary to get to this case. The other conclusions follow directly from the assumptions of the lemma. □

Theorem 7.4. *Correctness and complexity of Cat*

Let $l \in T$ and $r \in T$ be well formed and let s = Cat(l, r). Then it follows that rep(s) =rep(l) $\oplus$ rep(r), and Cat completes in time bounded by $O(\log_2(|\text{rep}(s)|))$

Proof. The cases where either sequence is empty are trivial, and they are ruled out prior to the call on CatRecurse, thus satisfying its precondition. In case1, the postconditions of CatRecurse directly satisfy the conclusion of the theorem. In case2, the postconditions of CatRecurse satisfy the preconditions of MakeNonleaf, whose postconditions in turn satisfy the conclusions of the theorem.

Each of the functions Height, Ct, and MakeNonleaf that is called in catRecurse has constant complexity. Likewise, each of the concatenations in Cat and in CatRecurse has at most $2d$ elements and is thus of constant bounded complexity. When CatRecurse is applied to nonleaf nodes whose maximum height is h, the recursive call is applied to nodes whose maximum height is $h-1$. Thus the algorithm terminates and has complexity $O(h) \leq \log_2(|\text{rep}(s)|)$, by the height bound theorem. □

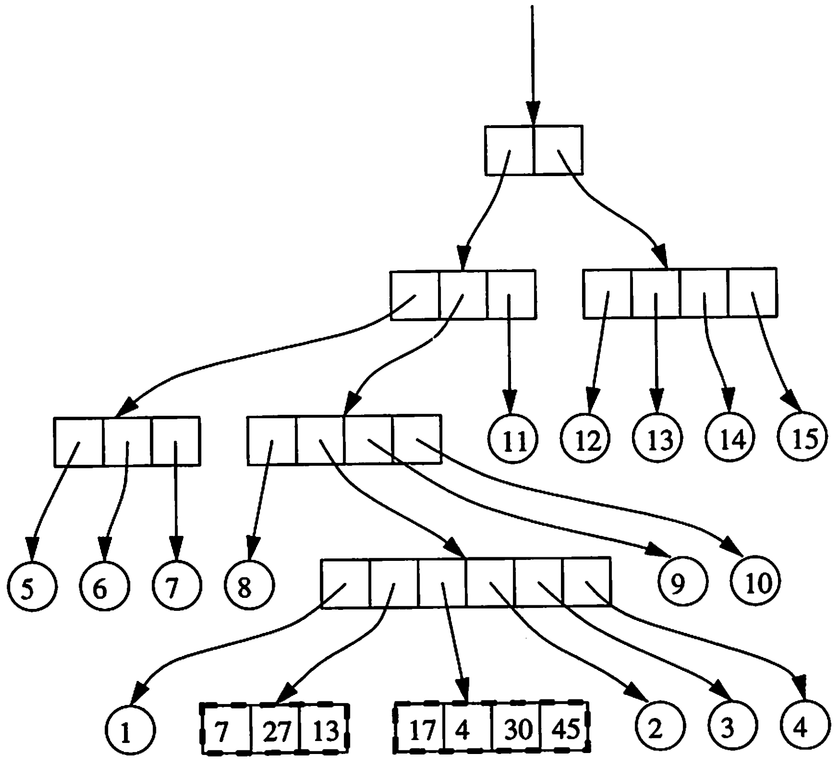

FIGURE 7.2. After concatenation

7.1. A Concatenation Example. We give an example of a concatenation, showing in full, only the nodes along the seam. In the example, $d = 6$. Fig. 7.1 shows two sequence-trees before the concatenation, with nodes of matching height aligned along the seam at the center. Fig. 7.2 shows the result. As before, reused nodes are shown boxed in heavy dashes. Height and CumChildCt fields of nonleaf nodes are omitted. The small numbers in circles represent nodes to either side of the seam without showing their contents. All such nodes are also reused.

At height 1, the two leaf nodes have collectively 7 elements, which cannot be repacked into one node, so they are reused intact in the result. At height 2, there are collectively 6 elements, so they are repacked into one new nonleaf node.

At height 3, there are initially 8 elements. Two of these point to nodes which are not reused in the result and which are both replaced by a single node. This leaves a total of 7 elements needed along the seam in the new sequence-tree at this height. The new elements are distributed 3 and 4 into the two new nodes.

At height 4, only the right operand sequence-tree has a node. The root pointer of the left operand is treated as a one-element fictitious node, which must be repacked with the elements from the right side of the seam. This requires a total of seven elements, two of which point to replacements for old nodes and the rest point to reused nodes to the right of the seam.

Finally, a new node at height 5 is needed to collect the two nodes of height 4. Thus the height of the result sequence-tree is one greater than the highest operand sequence-tree.

8. Slicing

Slicing of sequence-trees is performed bottom up by constructing two *cuts* through the operand sequence-tree along the left and right edges of what will become the result sequence-tree. At each height, the sequence-tree node is divided between the elements belonging to the slice and those outside the slice. The *node slice* of the divided node on its "sliceward" side needs to be included in the result sequence-tree.

The node slice could turn out to have only one element. If this happens, it is repacked with the adjacent node in the sliceward direction. This will give a total of at least 3 and at most $d + 1$ elements, which can always be packed into either one or two nodes. As an optimization to keep nodes more nearly full, the node slice and the adjacent node can also be repacked if they collectively will fit into one new node, even when the node slice has more than one element.

As with concatenation, the slice algorithm must start at the top of the operand sequence-tree, descend recursively, and do its reconstruction on the way back up. However, when slicing, the descending phase must determine, at each height, which child to descend to, using the starting and ending slice subscripts. The algorithm Sch is used to do this.

The descent is always into the sequence-subtree which contains the leftmost or rightmost slice element, respectively. This means that, at every height, the node slice will never be empty.

When computing a sufficiently broad slice at sufficiently low heights, the left and right cuts are separated by other sequence-tree nodes. Whenever at least two nodes separate the nodes containing the left and right cut, each side of the slice can be constructed independently, since each cut can involve repacking with at most one sliceward neighboring node.

At higher levels, the cuts merge. This can happen when three or fewer nodes are involved. Three nodes are involved at some level when a single node separates the nodes containing the left and right cuts. The left and right node slices could each be as short as one element and thus need to be repacked with the center node. The center node could have as few as two elements. Furthermore, at a nonleaf level, the level below could have replaced the leftmost two nodes with just one, and likewise for the rightmost two. At most, three full nodes could be needed to hold all the elements of the result sequence-tree at this level.

The two cuts can also be in adjacent nodes, leaving only two nodes involved. The cuts can also be in the same node. In all of these cases, the number of elements to be included in the result sequence-tree at this level can range from 1 to $3d$. When only one element exists, no construction is done at this level. Instead, the single pointer to a node at a lower level is just returned, eventually to be the result sequence-tree by itself. In this case, the result sequence-tree is lower than the operand sequence-tree.

In all other cases involving one, two, or three old nodes, the elements are repacked into one, two, or three new nodes, as needed. The pointers to these are returned to be collected by the next higher level. As long as the left and right cuts involve three or fewer nodes, the recursive descent through the operand sequence-tree must take care of both cuts in a single subroutine invocation, in order to handle these interactions between the two. Lower than this, independent handling of the left and right cuts can be done by independent recursion.

In all situations where elements are repacked into one, two, or three new nodes, it is always possible to satisfy the bounds on node degree. If the total number of elements to be repacked is three or less, one node will hold them, since d is at least 3. If the total is four or more, that is enough for two nodes, if required, to have two elements each. Larger total numbers of elements can always be divided among the minimum required number of nodes with at least two elements per node.

8.1. The Slicing algorithm.

The complete slicing algorithm is quite large. At a given height, the number of nodes involved can be one, two, three, or four or more. These four cases all require similar but distinct algorithms. There is, of course, a leaf and a nonleaf case for each of these. Furthermore, for each of nonleaf case, a distinct algorithm is needed for each of the possibilities of number of nodes involved at the next lower height, which be the same or larger. Finally, when the number of nodes involved is four or more, two distinct recursive algorithms are required for the left and right cuts.

Altogether, there are 10 differently coded leaf and 10 nonleaf cases. While all are conceptually similar, they differ sufficiently in detail that an attempt to merge them into fewer cases would doubtless result in a considerably more difficult algorithm, and probably no reduction in overall algorithm size. In one of the implementations that have been programmed, the concatenation routines require about 200 lines of code, whereas slicing requires over 1000.

In the interests of reasonable brevity, we present here, only one case, namely the recursive function for the right cut, when it is far enough removed so as not to interact with the left cut. Both leaf and nonleaf cases are included, since these can occur at any separation of cuts. We state and prove a lemma whose assumptions and conclusions are the needed preconditions and postconditions for the function.

Algorithm 8.1. *SliceRecurseRight*

```
FUNCTION SliceRecurseRight
  ( left , right : M ; i : ℵ ) : M × ( M ∪ Λ )
= IF i = Length ( right ) - 1
  THEN (* entire right node is included *) ( left , right )
  ELSE LET h := Height ( left )
    IN IF h = 1
      THEN (* leaf/singleton case *)
        LET new := left ⊕ right [ 0 ]
                 ⊕ ··· ⊕ right [ i - 1 ]
        IN MakeLeafPair ( new )
      ELSE (* h > 1, nonleaf case *)
        LET j := Sch ( right , i )
        , k := i - LeftCount ( right , j )
        , leftElems := Elems ( left )
        , rightElems := Elems ( right )
        , lct := len leftElems
        , ( leftContext , leftDesc )
            := IF j = 0
               THEN ( leftElems [ 0 ]
                      ⊕ ··· ⊕ leftElems [ lct - 2 ]
                    , leftElems [ lct - 1 ]
                    )
               ELSE ( leftElems ⊕ rightElems [ 0 ]
                      ⊕ ··· ⊕ rightElems [ j - 2 ]
                    , rightElems [ j - 1 ]
                    )
        , rightDesc := rightElems [ j ]
        , ( newLeft , newRight )
            := RecurseRight ( leftDesc , rightDesc , k )
        , new := IF newRight = Λ
            THEN leftContext ⊕ newLeft
            ELSE leftContext ⊕ newLeft ⊕ newRight
        IN MakeNonleafPair ( new )
```

Lemma 8.2. *Correctness and complexity of SliceRecurseRight*

Let $l_0 \in M$, and $r_0 \in M$ be well-formed, with height(l_0) = height(r_0). Let $0 \leq i <$ length(r_0). Further, let (l, r) = SliceRecurseRight(l_0, r_0, i). Then

(1) $l \in M$ is well-formed
(2) $r \in M \cup \Lambda$ is well-formed
(3) rep(l) $\oplus$ r = rep(l_0) $\oplus$ slice(rep(r_0), 0, i)
(4) Height(l) = Height(l_0)
(5) either $r = \Lambda$ or Height(r) = Height(l_0).

Proof. If i = length(right) − 1, all the desired conclusions follow directly. Otherwise, as usual, we will prove the lemma by induction on tree height.

Basis case, i.e. height = 1: This is the leaf case, $l_0 \in L$ and $r_0 \in L$. From the definition of L and the assumption $0 \leq i$, $3 \leq |\text{new}| < 2d$, so the preconditons of MakeLeafPair are satisfied, and the desired conclusions follow directly from the postconditions of MakeLeafPair and the assumptions of the lemma.

Induction step: Assume the lemma holds for sequence-trees of height $h - 1$, $h > 1$. For height h, this is a nonleaf case, i.e. Height(l_0) = Height(r_0) > 1. Prior to the recursive call, we have

$$\text{rep}(l_0) \oplus \text{slice}(\text{rep}(r_0), 0, i) = \\ \text{rep}(\text{leftContext}) \oplus \text{rep}(\text{leftDesc}) \oplus \text{slice}(\text{rep}(\text{rightDesc}), 0, k)$$

by construction of leftContext, leftDesc, rightDesc, and k. These also satisfy the precondition for the recursive call by their construction. Appealing to the inductive assumption, after new has been computed,

$$\text{rep}(\text{new}) = \text{rep}(\text{leftContext}) \oplus \text{rep}(\text{leftDesc}) \oplus \text{slice}(\text{rep}(\text{rightDesc}), 0, k)$$

From construction, we have $1 \leq |\text{leftContext}| \leq 2d - 2$, thus $2 \leq |\text{new}| \leq 2d$. By the inductive assumption, all elements of new have the same height, satisfying the preconditions of MakeNonleafPair, whose postconditions imply that rep(l) $\oplus$ rep(r) = rep(new). From the two forgoing conclusions about representations, this is equal to rep(l_0) $\oplus$ slice(rep(r_0), 0, i), which is conclusion (3) of the theorem. The postconditions of MakeNonleafPair directly imply the other conclusions of the lemma. □

The complete slicing algorithm includes four additional recursive functions. Function SliceRecurseLeft is symmetrical to SliceRecurseRight. It handles the left cut, accepting as parameters, the node in which the cut is to be found and the node to its right, along with the starting subscript of the beginning of the slice relative to the subtree it lies in.

Function SliceRecurse3 handles both cuts, when there is one additional node between the one containing the left cut and that containing the right cut. It accepts these three nodes and the subscripts of the two cuts and

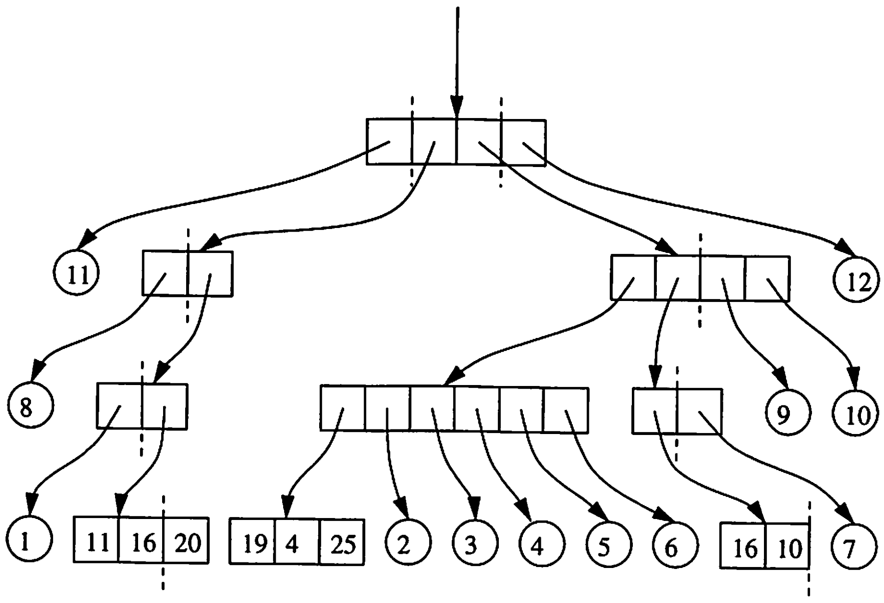

FIGURE 8.1. Before slicing

returns as many as three reconstructed nodes, two of which could be absent, as represented by Λ. To construct its results, it uses a support function MakeNonleafTriple, which is very similar to MakeNonleafPair, but accepts a sequence in $M^{2,3d}$ and can build as many as three nodes. It calls SliceRecurseLeft and SliceRecurseRight.

Function SliceRecurse2 accepts two adjacent subtrees and two subscripts, where the two cuts lie in the two trees. It returns one or two subtrees. It can make a recursive call on itself, on SliceRecurse3, or SliceRecurseLeft and SliceRecurseRight.

Function SliceRecurse1 accepts a single tree in which the two cuts are known to lie, along with the usual two subscripts. It returns one tree, which is the desired slice. It may make a recursive call on itself, on SliceRecurse2, SliceRecurse3, or both SliceRecurseLeft and SliceRecurseRight.

Finally, there is a top level function which checks the initial subscript bounds and calls SliceRecurse1. A complete implementation of all the sequence-tree algorithms is available, in both Modula-3 and Ada at:
`http://www.cs.twsu.edu/~rodney/`

8.2. **A Slicing Example.** Here, we give an example of slice construction, showing only relevant nodes along the cuts. Fig. 8.1 is the operand sequence-tree and Fig. 8.2 is the result slice sequence-tree. The notation is

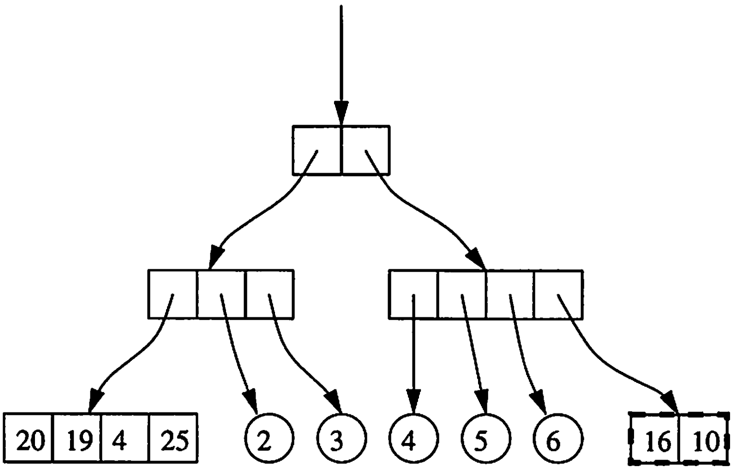

FIGURE 8.2. After slicing

the same as in Fig. 7.1, with the extension that dashed lines through nodes of the original sequence-tree are used to show the location of the left and right cuts.

At height one, the two cuts are independent. On the left, the node slice of the leftmost node shown in full contains one element whose value is 20. This is repacked with the 3 elements 19, 4, and 25 of the next rightward node to give a new node with 4 elements. On the right cut at this same level, the entire node containing element values 16 and 10 is reused.

At height 2, three nodes are involved in the slice. At the left cut, two elements have been replaced by one new element, returned from below. All the rest of the elements involved are reused. This gives a total of 7 elements for the result sequence-tree at this level. These are repacked into two new nodes, whose pointers are returned to height 3.

At height 3, only two nodes are involved. The level below returns two new node pointers to replace the three that were passed down to it. These are packed into a single new node.

Finally, at height 4, only one node is involved. Two of its elements are passed down and one new element is returned. Since the one node pointer does not require a node at this level, it is just returned from level 4 unchanged, to become the entire result sequence-tree. The result height is thus one less than that of the operand.

8.3. Complexity of Slicing. The slice algorithm proceeds down two cuts, using the same technique as single element access. This is $O(\log_2(|s|))$, where $s = \text{rep}(t)$, and t is the operand sequence. During its bottom-up reconstruction, at most two cuts times two nodes or $4d$ total elements are visited at each level. This is again $O(\log_2(|s|))$, which is the complexity of the overall slice algorithm.

9. Discussion

In the implementations done, whenever several elements are repacked into two or three nodes, they are distributed among the nodes as nearly equally as possible. This is not essential to the correctness or complexity bounds of the algorithms, but it attempts to minimize skewing of the distribution of node sizes. This should presumably improve the actual overall space and time complexity.

The actual implementation of sequence-tree operations turns out to be complex in handling the various cases. Slicing near the top where the cuts are merging is particularly complex. A fairly pure implementation in Modula-3, containing single element access and assignment, concatenation, slicing, and iteration over sequences consumes about 2000 lines of source code, using a rather liberal formatting style. Additional code to dump and consistency-check sequence-trees is not counted in this number, nor is garbage collection itself nor support therefor (e.g. maintenance of reference counts) included.

Variations on the operations are possible, using the same data structure and invariants. In a given application, where the pattern of sequence operations is known, it is often sensible to make some of the operations lazy. This is particularly likely to apply to slicing.

For sequence-trees, making some operations lazy and combining their eventual evaluation with other operations can lead further to specialized operations which have better constant-factor efficiency, keep nodes more nearly full, and produce less garbage. The algorithmic complexity is, of course, correspondingly increased. Single-element insertion and deletion are relatively simple examples.

In the semantic editor for which sequence-trees were initially developed, slices are only used as intermediate results, which are always then concatenated. Furthermore, the concatenations are, in general, multi-operand. This implementation uses the data structure definition and the single element access and assignment operations as given here, but has a composite operation which simultaneously slices and then concatenates the intermediate slice to another sequence. This avoids the significant work entailed by computing autonomous slices, because a one-element node slice need not be repacked with a sliceward node. Instead, it will be repacked with other elements on the other side during the following concatenation.

Additionally, since this implementation assumes there will be multiple concatenations, it temporarily keeps the state of the result sequence-tree along one seam in pseudo-nodes. These contain the same information as real nodes, but are preallocated and always have enough space for d elements, even when not necessarily all elements are used. Since they are never shared, they need not be immutable and can have additional elements packed in during later concatenations. This reduces the production of garbage and aids in fully packing nodes near the top of the result tree. When all concatenations of a single result sequence are complete, the pseudo-nodes along the seam are copied into real, heap-allocated, exact-size nodes.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms* Addison-Wesley, Reading MA, 1974.

[2] R. J. Baron and L. G. Shapiro. *Data Structures and their Implementation*. Van Nostrand Reinhold, New York, NY 1980.

[3] M. R. Brown and R. E. Tarjan. Design and analysis of Data Structures for Representing Sorted Lists. *SIAM J. Comput.*, 9,3 (Aug. 80), pp. 594–614.

[4] N. Dale and H. M. Walker. *Abstract Data Types*. D. C. Heath and Co., Lexington, MA, 1996.

[5] M. Dagenais, ftp://ftp.vlsi.polymtl.ca/pub/m3/sequences-0.0.tar.gz

[6] D. P. Dobkin and J. I. Munro. Efficient Uses of the Past. *Proceedings of 21st Annual IEEE Symposium on Foundations of Computer Science*, Syracuse, NY Oct. 13–16, 1980, pp. 200-206.

[7] L. J. Guibas and R. Sedgewick. A Dichromatic Framework for Balanced Trees. *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science*, Ann Arbor MI, Oct. 16–18, 1978, 8-21.

[8] E. Horowitz and S. Sahni. *Fundamentals of Data Structures*. Computer Science Press, Potomac MD, 1976.

[9] P. Helman and R. Veroff *Intermediate Problem Solving with Data Structures*. Benjamin/Cummings, Redwood City, CA, 1991.

[10] S. Huddleston and K. Mehlhorn. A New Data Structure for Representing Sorted Lists. *Acta Informatica*, 17,2, (Jun. 82), pp. 157–184.

[11] E. W. Myers, Efficient Applicative Data Types. *Conference Record, 11th Annual ACM Symposium on Principles of Programming Languages*, Salt Lake City, UT, Jan. 15–14, 1984, pp. 66–75.

[12] J. Nievergelt and E. M. Reingold. Binary Search Trees of Bounded Balance. *SIAM J. Comput.* 2,1 (Mar. 73), pp. 33–43.

[13] T. Reps, T. Teitelbaum, and A. Demers. Incremental Context-Dependent Analysis for Language-Based Editors. *ACM Transactions on Programming Languages and Systems*, 5,3 (Jul. 83), pp. 449–477.

[14] N. Sarnak and R. E Tarjan. Planar Point Location Using Persistent Search Trees. *ACM Communications*, 29,7 (Jul. 86), 669-679.