# Contents

## 3.5 Abstract Data Types for Immutable Data

The design of a data structure is normally tightly coupled with the ADT being implemented. The property of immutability permits the development of a general technique for implementing an ADT. The technique has not previously been explored in the context of an imperative programming language. The Canonical Binary Tree can be made to conform to many different ADTs by specifying a specialising function as a first order parameter. Functions acting on the Canonical Binary Tree, including those supporting concurrent execution, can be implemented independent of the ADT.

The main contribution of this section is the development of an Immutable Data Structure that separates the concerns of the structure from those of the ADT to which it conforms. This section focuses on techniques for specialising the Canonical Binary Tree so that a mechanism to allow concurrent access can be implemented independent of the ADT.

### 3.5.1 Priority Queue

A priority queue associates a priority with a data value so that the value associated with the highest priority can be recovered. Priority queues are used to schedule operating system tasks and to solve the selection problem, which is to return the $k$th largest element from a set of elements.

A priority queue has a $Push()$ function to insert a value with an associated priority into the structure. It has a $Top()$ function that returns the value with the highest priority and a $Pop()$ function that removes that value. It is conventional to regard low numbers as high priorities.

Hinze describes an implementation of a purely functional priority queue based on a min-tree [HP05]. The min-tree is a type of tournament tree in which the annotation of a leaf is the priority and the annotation of a node is the minimum annotation of its children. This property causes the annotation of the root node to be equal to the lowest priority of any leaf. A path from the root to the leaf with the highest priority is found by examining the annotation of the root node and then repeatedly choosing the child node with matching priority until a leaf is reached.

Figure 3.1 illustrates an example of a min-tree.

The min-tree corresponds to a mathematical expression in which the minimum

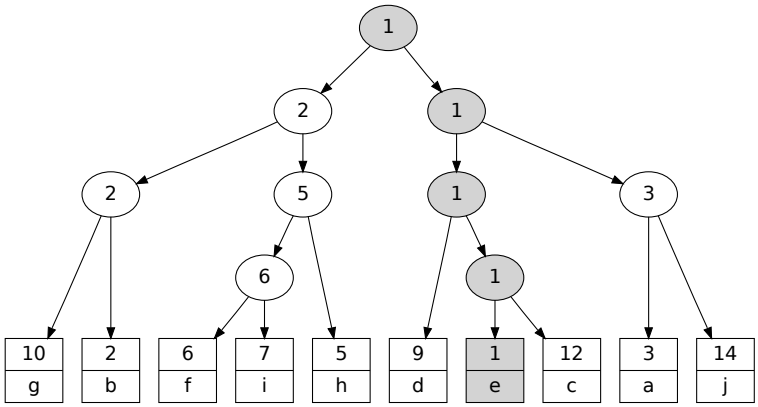
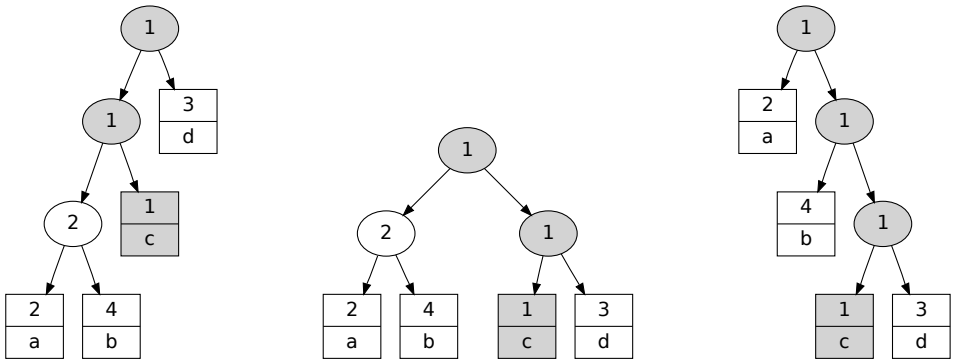
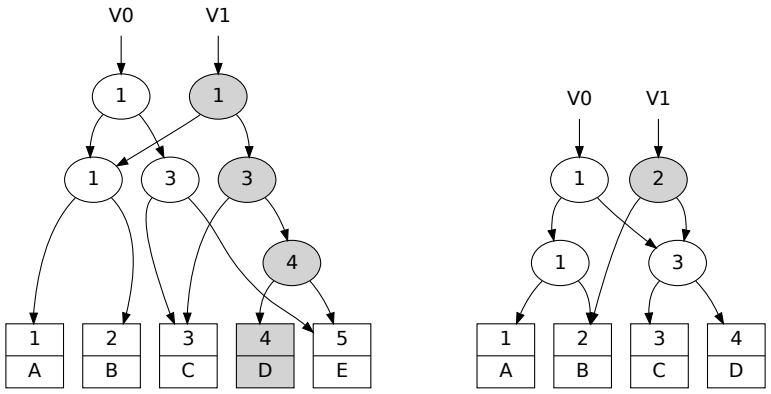



Figure 3.1: **Example Min-tree** containing the priority value pairs $\{1 \mapsto e, 2 \mapsto b, 3 \mapsto a, 5 \mapsto h, 6 \mapsto f, 7 \mapsto i, 9 \mapsto d, 10 \mapsto g, 12 \mapsto c, 14 \mapsto j\}$. The shaded vertices illustrate the path to the value with the highest priority.




(a) (b) (c)

Figure 3.2: **Associativity property of a min-tree.** Min-trees with different topologies maintain the property that the root node is annotated with the minimum annotation of any leaf. The shaded vertices illustrate the path to the highest priority element.




(a) (b)

Figure 3.3: **Insertion and removal of an element in a min-tree.**
(a) Insertion of an element into an immutable min-tree. Version V0 contains the priority value pairs $\{1 \mapsto A, 2 \mapsto B, 3 \mapsto C, 5 \mapsto E\}$. The operation $Push(4 \mapsto D)$ creates version V1 containing the priority value pairs $\{1 \mapsto A, 2 \mapsto B, 3 \mapsto C, 4 \mapsto D, 5 \mapsto E\}$. The path created by the operation is shaded.
(b) Removal of an element from an immutable min-tree. Version V0 contains the priority value pairs $\{1 \mapsto A, 2 \mapsto B, 3 \mapsto C, 4 \mapsto D\}$. The operation $Pop()$ creates version V1 containing the priority value pairs $\{2 \mapsto B, 3 \mapsto C, 4 \mapsto D\}$. The path created by the operation is shaded.

function is applied to the priorities. The minimum function is both associative and commutative so the min-tree maintains the property that the annotation of a node is equal to the minimum priority of any leaf in the subtree that it suspends, regardless of the topology of that subtree.

Figure 3.2 illustrates the associativity property of the minimum function.

The $Push()$ function inserts a value into the min-tree by creating a new leaf containing the value and annotated by the priority. A new path from the root to this leaf is created by path copying. The annotation of each node on the path is set to be the minimum of the annotations of its children. Path copying creates an entirely new path so the annotations of existing nodes are unaffected by the operation. The $Push()$ function can insert a leaf anywhere in the tree because the minimum function is commutative.

The $Pop()$ function removes the value with the highest priority from the immutable min-tree by creating a new path which makes the leaf with the highest priority unreachable. The root node is annotated with the next highest priority.

Figure 3.3 illustrates the insertion and removal of an element from a min-tree.

Figure 3.4 illustrates the growth of an immutable min-tree. Successive leaves are added through a process of path copying. The properties of the min-tree are preserved by each version.

The min-tree requires that the children of a node are examined when determining the path. If the path could be determined without accessing the children then the number of nodes accessed when traversing a path would be approximately halved.

In the context of concurrent execution the benefit of determining the path without accessing the children is significant because nodes that are read while traversing the path must be recorded to ensure correct concurrent execution. It is therefore beneficial to restrict node access to those nodes actually on the path.

The min-tree requires that both the comparison and the annotator function are supplied as specialising functions. The annotation of the root node is followed to the leaf. This requires a special comparison operation to reach the highest priority element because the value of the annotation of the root node must be retained while following the path. If the path could be determined without specialising the comparison operation then the amount of information required to describe the data structure would be reduced.

The min-tree does not specify a representation of the empty priority queue. If a representation of the empty priority queue were specified it would be possible to distinguish an empty priority queue from a non-existent queue.

In a typical priority queue implementation the $Pop()$ function behaves differently when removing the last remaining element in a data structure because the data structure is subsequently empty. In the concurrent execution environment the status of a data structure between function calls is unknown so it is necessary that the data structure represents and includes checks for an empty queue in access function.

The functions $Top()$, $Push()$ and $Pop()$ are specific to the priority queue ADT. If these functions could be specified as adaptations of the access functions of the Canonical Binary Tree then it would be possible to abstract the priority queue ADT from the data structure that implements it.

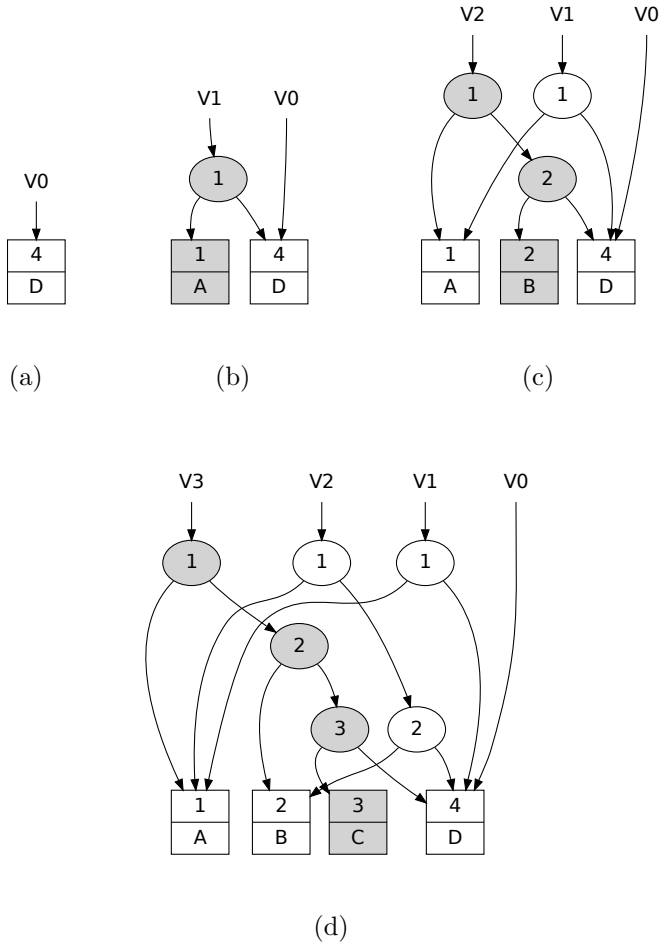



Figure 3.4: **Animation showing the growth of a min-tree** through a series of insertions. A new version of the data structure is created by each operation. In each case the path created by the operation is shaded.
(a) Initial data structure containing the priority value pair $4 \mapsto D$.
(b) After $Push(1 \mapsto A)$
(c) After $Push(2 \mapsto B)$
(d) After $Push(3 \mapsto C)$.

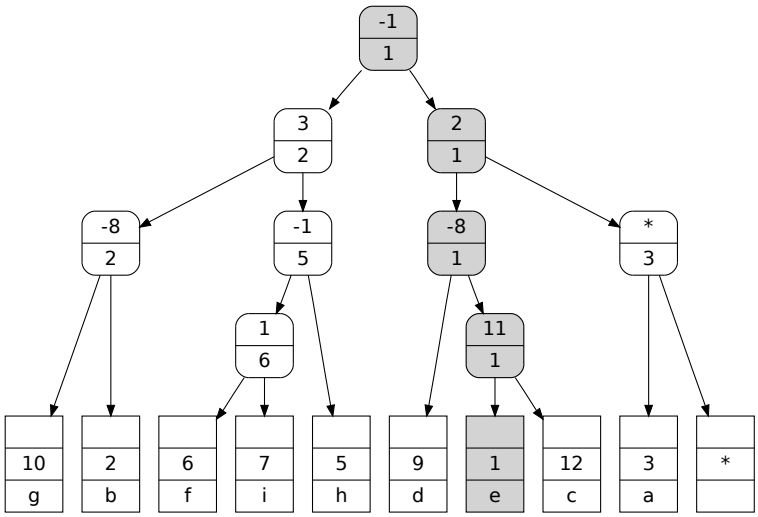



Figure 3.5: **Example Directed min-tree** containing the priority value pairs $\{1 \mapsto e, 2 \mapsto b, 3 \mapsto a, 5 \mapsto h, 6 \mapsto f, 7 \mapsto i, 9 \mapsto d, 10 \mapsto g, 12 \mapsto c\}$. The shaded vertices illustrate the path to the highest priority element. The sentinel is the right-most leaf of the tree. The first annotation is shown above the second annotation.

### 3.5.2 Directed min-tree

A new data structure, the directed min-tree, implements the priority queue ADT and addresses the shortcomings of the min-tree.

The min-tree suffers from the problem that the annotations of both of the children must be examined to determine the path. The directed min-tree solves this problem by regarding the annotation as a pair of values. One value contains the minimum value of the child annotations and the other contains an indicator as to whether the left or right child of a node has a lower annotation value.

Figure 3.5 illustrates the annotations of a directed min-tree. The annotation pair contains two values that we call the first and second annotations of the node. In the figure the first annotation is shown above the second annotation. The first annotation is calculated by subtracting the second annotation of the left child from the second annotation of the right child. The second annotation is the minimum of the second annotations of the children. The first annotation of a leaf is not used and its second annotation is the priority associated with the application value.

Only the first annotation of a node is examined when traversing the path. This annotation indicates which child has the minimum second annotation. The path to the leaf with the highest priority can be found by comparing the first annotation of each node with zero. If it is greater than zero then the left child is on the path, so to determine the path it is only necessary to examine the annotations of nodes on the path.

The min-tree suffers from the problem that both the comparison operation and the annotator function must be supplied as specialising functions, whereas the directed min-tree requires only that the annotator function be specified. The comparison function is regarded as a feature of the Canonical Binary Tree common to all ADTs.

The comparison function and the path determination process are the same regardless of the ADT being implemented, so the *query*() function is ADT agnostic. For example, the *Top*() function is implemented as a Canonical Binary Tree *query*() function with an access parameter of zero. Path determination is a common feature of the *query*(), *insert*() and *delete*() functions so the implementation of each function is simplified by making path determination ADT agnostic.

The min-tree suffers from the problem that the annotation of the root node must be an argument to the comparison function for every node on the path, whereas the directed min-tree does not treat the root node as special and does not require an annotation to be retained while determining the path.

The min-tree suffers from the problem that it does not specify a representation of the empty priority queue, whereas the directed min-tree contains a sentinel that can be used to distinguish an empty data structure from a non-existent data structure. The sentinel is annotated in such a way that it cannot be removed from the tree.

The access parameter of the *insert*() function identifies a leaf in the data structure. When a new leaf is inserted to the min-tree it can be inserted either to the left or the right of this leaf because the minimum function is commutative so it does not matter on which side of the path the insertion takes place. However, the Canonical Binary Tree requires that the sentinel is always the right most leaf. To ensure this, the insert function always inserts a new leaf to the left of the path identified by the access parameter. When the tree is created the sentinel is the only leaf and the *insert*() function always inserts leaves to the left of the path,

| Canonical Binary Tree specialisation | |
|---|---|
| $annotator(< a, b >, < c, d >)$ | $< d - b, min(b, d) >$ |
| $identity$ | $<, \infty >$ |

| API function | Canonical Binary Tree access function |
|---|---|
| $Push(priority)$ | $insert(priority)$ |
| $Pop()$ | $delete(0)$ |
| $Top()$ | $query(0)$ |

Table 3.1: **Directed min-tree implementation.** The Canonical Binary Tree can be specialised implement a directed min-tree and its access functions can be adapted to present a priority queue ADT to the application.

so the sentinel will always remain the right-most leaf of the tree.

The sentinel must be annotated in such a way that the left child of its parent is always chosen by the $query()$ and $delete()$ functions because the sentinel is unreachable by $query()$ and cannot be removed by $delete()$. The second annotation of the sentinel is infinity which causes its parent to have a first annotation value of infinity so a path through the directed min-tree will include the sentinel only when the tree is empty. In practice, the sentinel is annotated with the highest value of the data type of the annotation.

The min-tree suffers from the problem that the ADT cannot be completely abstracted from the data structure that implements it, whereas the directed min-tree can be implemented by specifying access arguments to adapt the functions of the Canonical Binary Tree.

Table 3.1 contains all of the information needed to specialise the Canonical Binary Tree so that it implements the priority queue ADT. The annotator function returns the annotation of a node given the annotations of its children. The identity is the annotation of the sentinel. The $Push()$ function is implemented by the $insert()$ function of the Canonical Binary Tree. The $Pop()$ function is implemented by the $delete()$ function, The value with the highest priority will always be found by specifying an access argument of zero as the access parameter of the $delete()$ function. Similarly, the $Top()$ function is implemented by a $query()$ with an access argument of zero.

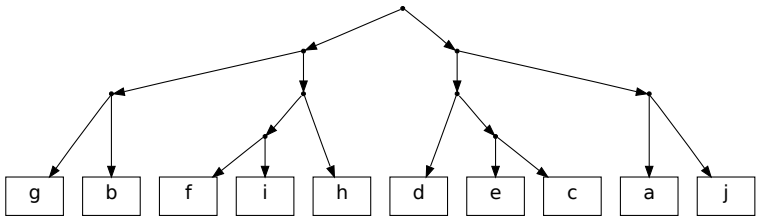



Figure 3.6: **Example Deque** containing the values $\{g, b, f, i, h, d, e, c, a, j\}$.

### 3.5.3 Deque

A deque data structure contains an ordered list of elements and only permits access to those elements at either end of the list. The functionality of the data structure can be further restricted to implement a queue or stack.

A deque is regarded as having a front and a back. The $Push\_front()$ function inserts a value onto the front of the deque. The $Front()$ function returns that value. The $Pop\_front()$ function removes the value at the front of the deque. The corresponding functions $Push\_back()$, $Back()$ and $Pop\_back()$ affect the back of the deque.

Hinze describes an implementation of an immutable deque based on the ordering of leaves of a binary tree [HP05].

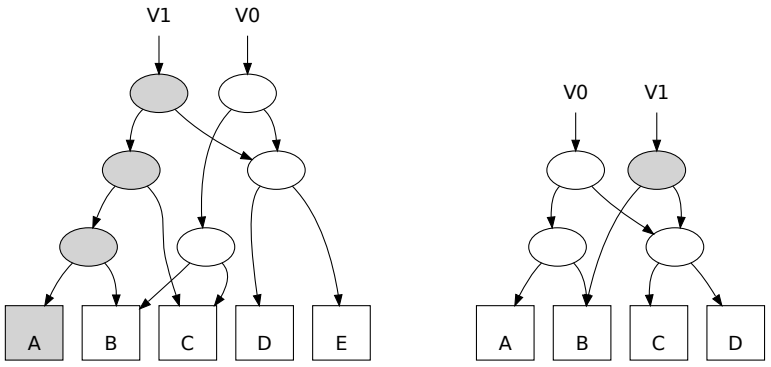
Figure 3.6 illustrates an example of a deque.

The vertices of the tree are not annotated. The front of the deque is found by choosing the left child of each node starting from the root node. The order of the leaves is preserved.

Figure 3.7 illustrates the insertion and removal of an element in a deque.

Figure 3.8 illustrates the growth of the immutable deque. Successive leaves are added through a process of path copying.

The deque corresponds to an expression in which the list concatenation function is applied to the values. The concatenation function is associative so the deque maintains the property that the annotation of a node is equal to the concatenation of the values of the leaves in the subtree that it suspends. It is not necessary to annotate the nodes with the value of the concatenation. List concatenation is not commutative so the order of leaves must be maintained during any transformation of the tree.




(a) (b)

Figure 3.7: **Insertion and removal of an element in a deque.**
(a) Insertion of an element into an immutable deque. Version V0 contains the values $\{B, C, D, E\}$. The operation $Push\_front(A)$ creates version V1 containing the values $\{A, B, C, D, E\}$. The path created by the operation is shaded.
(b) Removal of an element from an immutable deque. Version V0 contains the values $\{A, B, C, D\}$. The operation $Pop\_front()$ creates version V1 containing the values $\{B, C, D\}$. The path created by the operation is shaded.

Special comparison functions are required to reach the front and back of the deque. One of the comparison functions creates a path to the front of the queue by always selecting the left child. The other comparison function accesses the back of the queue.

This deque suffers from some of the same shortcomings as the priority queue, it requires the implementation of access functions that are specific to the deque ADT, it requires multiple comparison functions and it does not distinguish an empty deque from a non-existent deque.

### 3.5.4 Directed deque

A new data structure, the directed deque, addresses the shortcomings of the deque. It supports a sentinel and fully abstracts the ADT implementation from the functions of the Canonical Binary Tree.

The nodes are annotated with a pair formed from the second annotation of the child on the right and the second annotation of the child on the left. The first annotation of a leaf is not used and the second annotation is zero. The second

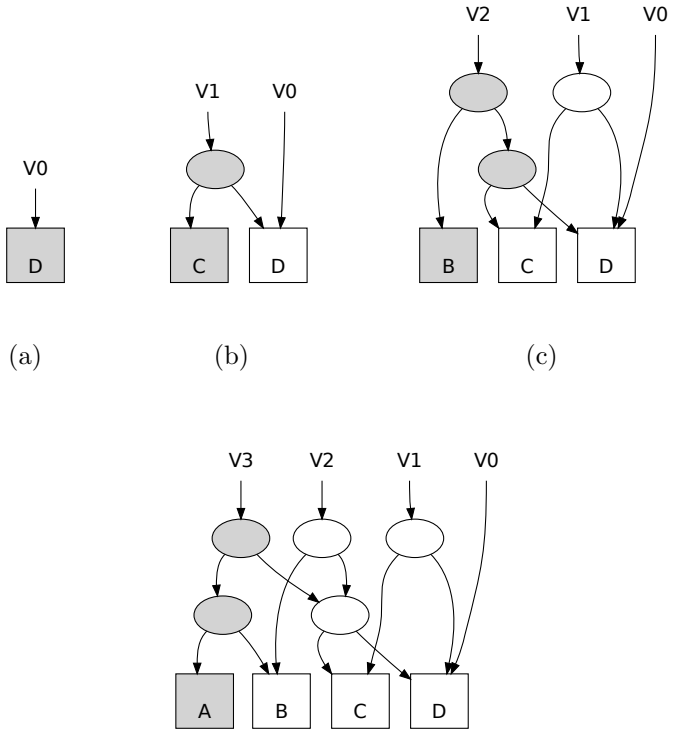



Figure 3.8: **Animation showing the growth of an immutable deque** through a series of insertions. New versions of the data structure are created by each operation. In each case the path created by the operation is shaded.
(a) Initial deque.
(b) After $Push\_front(C)$
(c) After $Push\_front(B)$
(d) After $Push\_front(A)$

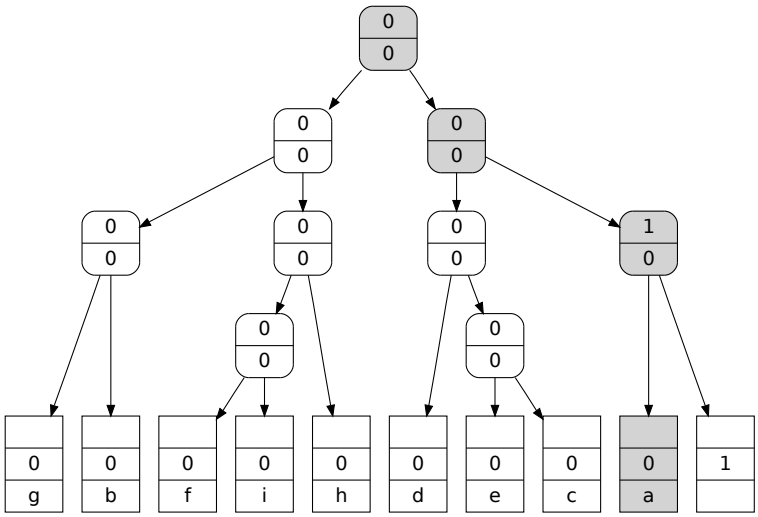



Figure 3.9: **Example Directed deque** containing the values $\{g, b, f, i, h, d, e, c, a\}$. The shaded vertices illustrate the path to the back of the queue. The sentinel is the right-most leaf of the tree. The first annotation is shown above the second annotation.

annotation of the sentinel is one.

Figure 3.9 illustrates an example of a Directed deque.

The sentinel is annotated in such a way that it cannot be removed from the tree and leaves cannot be inserted to the right of the sentinel. Using the annotation scheme three leaves are reachable. The left-most leaf, the sentinel and the leaf to the left of the sentinel.

The $Push\_front()$, $Front()$ and $Pop\_front()$ functions are implemented by the Canonical Binary Tree functions $insert()$, $query()$ and $delete()$ each called with an access parameter of zero which causes the path to the front of the queue to be selected. The $Push\_back()$ function is implemented by the $insert()$ function with an access parameter of infinity, which causes a path to the sentinel to be selected. Insertion takes place to the left of the sentinel which causes an element to be added to the back of the queue. The $Back()$ and $Pop\_back()$ functions are implemented by the $query()$ and $delete()$ functions with an access parameter of one which causes a path to the back of the queue to be selected.

Table 3.2 contains all of the information needed to specialise the Canonical Binary Tree so that it implements the directed deque ADT.

| Canonical Binary Tree specialisation | |
|---|---|
| $annotator(< a, b >, < c, d >)$ | $< d, b >$ |
| $identity$ | $<, 1 >$ |

| API function | Canonical Binary Tree access function |
|---|---|
| $Push\_front()$ | $insert(0)$ |
| $Pop\_front()$ | $delete(0)$ |
| $Front()$ | $query(0)$ |
| $Push\_back()$ | $insert(\infty)$ |
| $Pop\_back()$ | $delete(1)$ |
| $Back()$ | $query(1)$ |

Table 3.2: **Directed deque implementation.** The Canonical Binary Tree can be specialised to implement a Directed deque and its access functions can be adapted to present a deque ADT to the application.

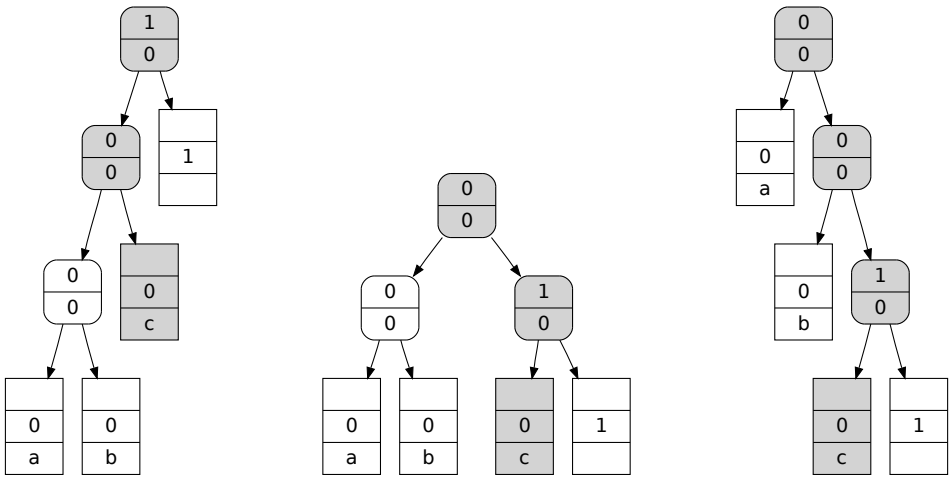



Figure 3.10: **Associativity property of a directed deque.** Directed deques with different topologies maintain the order of their leaves. The path to the back of the queue is shaded.

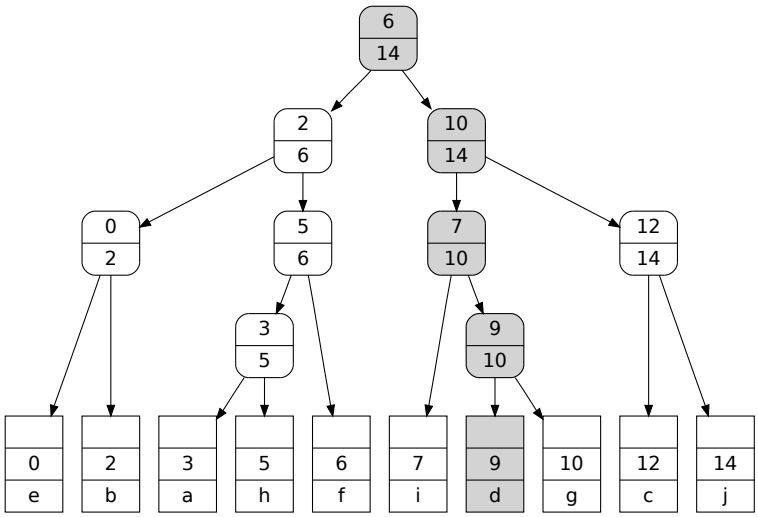



Figure 3.11: **Example interval tree** containing the key-value pairs $\{0 \mapsto e, 2 \mapsto b, 3 \mapsto a, 5 \mapsto h, 6 \mapsto f, 7 \mapsto i, 9 \mapsto d, 10 \mapsto g, 12 \mapsto c, 14 \mapsto j\ \}$. The shaded path illustrates the mapping $9 \mapsto d$.

Figure 3.10 illustrates the associativity property of the Directed deque. The associativity property allows the topology of the data structure to be modified without affecting the functionality provided by the ADT.

### 3.5.5 Map

A map is a sorted associative data structure that provides access to a set of key-value pairs. It also supports in-order traversal of leaves in sorted order. The functionality that the map ADT provides is similar to that of a C++ STL map [Jos99].

A map has an *Insert*() function that inserts a key-value pair, a *Query*() function that retrieves an application value given its key and a *Remove*() function that removes the key-value pair from the map.

Hinze describes an implementation of an immutable map using an interval tree [HP05].

Figure 3.11 illustrates an interval tree.

A interval tree corresponds to a mathematical expression in which the maximum and minimum functions are applied to the annotations. The first annotation

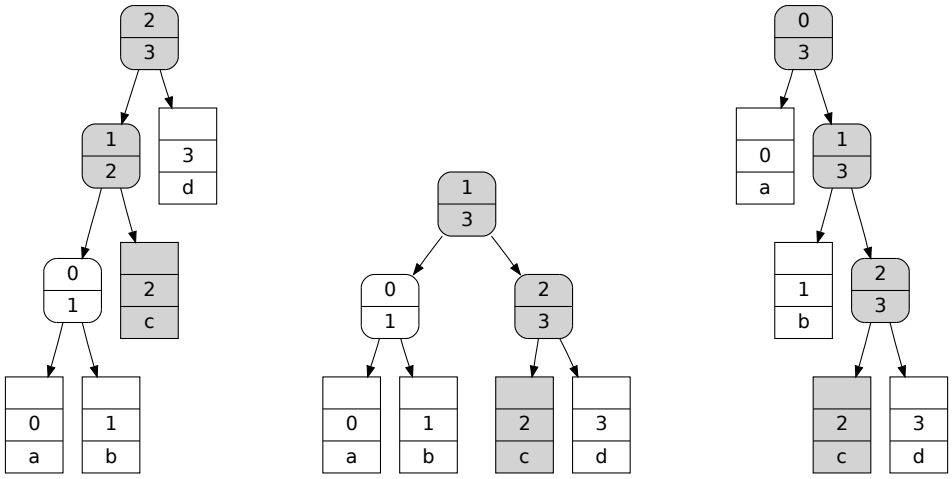


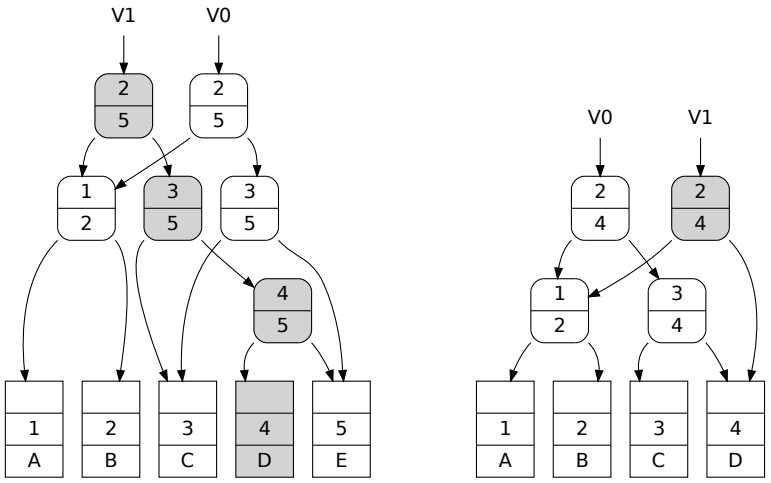

Figure 3.12: **Associativity property of an interval tree.** Interval trees with different topologies maintain the property that the first annotation of a node is the highest first annotation in the subtree suspended on the left and the second annotation is the highest first annotation in the subtree suspended on the right.

of a node is the minimum of the second annotations of its children. The second annotation of a node is the maximum of the second annotations of its children. The minimum and maximum functions are associative, so during topological transformations the interval tree maintains the property that the first annotation of a node is the maximum key in the sub-tree suspended by its left child and the second annotation of a node is the maximum key in the sub-tree suspended by its right child.

Figure 3.12 illustrates the associativity property of the interval tree.

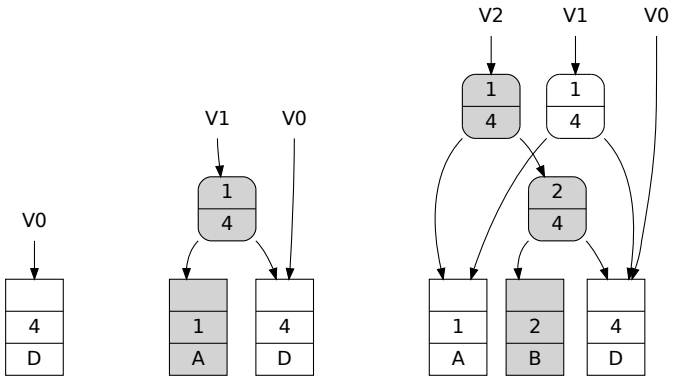
The key is the access parameter for the functions of the data structure. A path from the root to a leaf with a given key is found by repeatedly checking for a leaf and then comparing the key to the first annotation of the node. If the key is greater than the first annotation then the right child of the node is on the path. If a leaf with a given key is not present in the interval tree then a leaf with a different key will be found. It is not necessary to access the children of a node in order to determine the path.

Figure 3.13 illustrates the insertion and removal of an element in an interval tree.




(a) (b)

Figure 3.13: **Insertion and removal of an element in an interval tree.**
(a) Insertion of an element into an immutable interval tree. Version V0 contains the key-value pairs $\{1 \mapsto A, 2 \mapsto B, 3 \mapsto C, 5 \mapsto E\}$. The operation $Insert(4 \mapsto D)$ creates version V1 containing the key-value pairs $\{1 \mapsto A, 2 \mapsto B, 3 \mapsto C, 4 \mapsto D, 5 \mapsto E\}$. The path created by the operation is shaded.
(b) Removal of an element from an immutable interval tree. Version V0 contains the key-value pairs $\{1 \mapsto A, 2 \mapsto B, 3 \mapsto C, 4 \mapsto D\ \}$. The operation $Remove(3)$ creates version V1 containing the key-value pairs $\{1 \mapsto A, 2 \mapsto B, 4 \mapsto D\}$. The path created by the operation is shaded.



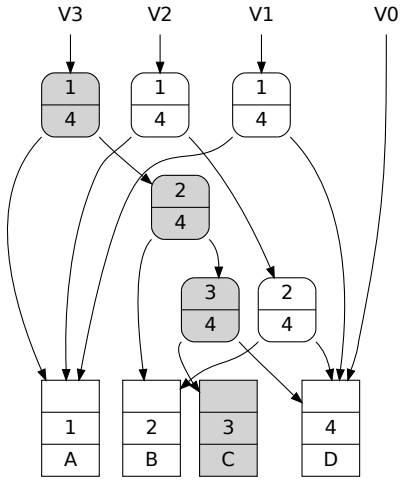




Figure 3.14: **Animation showing the growth of an interval tree** through a series of insertions. New versions of the data structure are created by each operation. In each case the path created by the operation is shaded.
(a) The initial data structure containing the key-value pair $4 \mapsto D$.
(b) After $Insert(1 \mapsto A)$
(c) After $Insert(2 \mapsto B)$
(d) After $Insert(3 \mapsto C)$

| Canonical Binary Tree specialisation | |
|---|---|
| $annotator(< a, b >, < c, d >)$ | $< min(b, d), max(b, d) >$ |
| $identity$ | $<, \infty >$ |

| API function | Canonical Binary Tree access function |
|---|---|
| $Insert(key)$ | $insert(key)$ |
| $Remove(key)$ | $delete(key)$ |
| $Query(key)$ | $query(key)$ |

Table 3.3: **Map implementation.** The Canonical Binary Tree can be specialised to implement an interval tree and its access functions can be adapted to present a map ADT to the application.

Figure 3.14 illustrates the growth of the immutable interval tree.

The interval tree suffers from the problem that it does not specify a representation of the empty map.

### 3.5.6 Interval tree with sentinel

The Canonical Binary Tree can be adapted to implement an interval tree with a sentinel. The first annotation of the sentinel is not used and the second annotation is infinity. In practice, the sentinel is annotated with the maximum value of its data type.

Table 3.3 contains all of the information needed to specialise the Canonical Binary Tree so that it implements the map ADT.

The $Query()$ function returns a value for every possible value of the access parameter, even when the access parameter does not match a key. This is not the behaviour typically expected of a map. The ADT wrapper functions can implement checks to ensure that the value retrieved by a query corresponds to the key and that duplicate keys are handled appropriately.

A $Query()$ function that checks that value returned corresponds to the key supplied as an access parameter can be implemented by storing the value of the key as part of the application value. In a concurrent execution environment the annotation of a leaf cannot be used for this purpose as it is regarded as structural information and is not accessible through the functions of the Canonical Binary Tree.

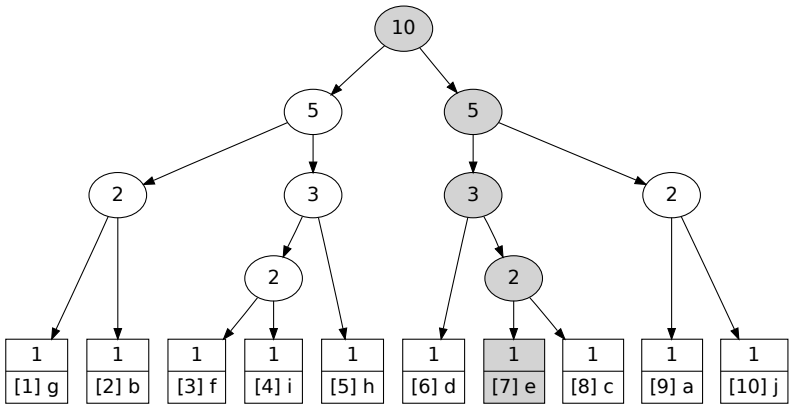



Figure 3.15: **Example Sequence tree** containing the values $\{[1]g, [2]b, [3]f, [4]i, [5]h, [6]d, [7]e, [8]c, [9]a, [10]j\}$

A set ADT can be implemented by an interval tree that does not permit duplicate values. To implement the set ADT the $Insert()$ function should call $query()$ to ensure the uniqueness of a value before calling $insert()$. In a concurrent execution environment the Canonical Binary Tree functions should be referentially transparent so the $insert()$ function, which alters the Canonical Binary Tree, cannot give any indication of success.

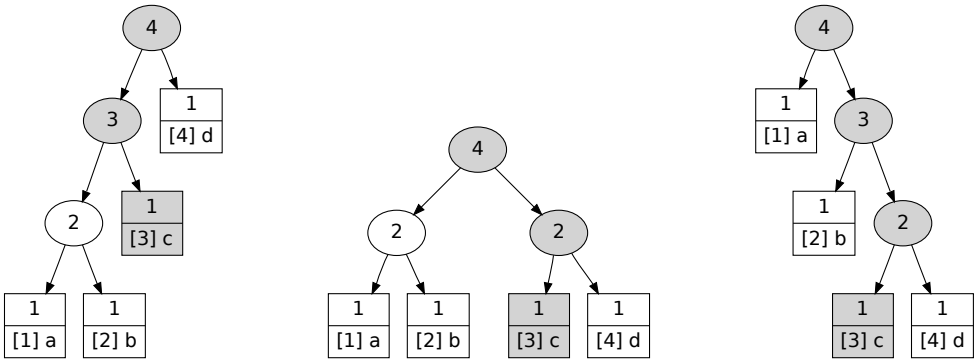
### 3.5.7 Vector

A vector is an ordered set of values that supports random access based on ordinal number. The vector ADT provides a similar set of functions to the deque ADT in addition to random access to ordinals within the sequence. The functionality that the vector ADT provides is similar to that of a C++ STL map [Jos99].

The $Insert()$ function inserts an ordinal value pair into the vector. The $Query()$ function is supplied with an ordinal as the access parameter. The function returns the value associated with the ordinal. The $Remove()$ function deletes an ordinal value pair from the data structure. An in-order traversal of the vector returns values in the order given by their ordinal number.

Hinze describes an implementation of an immutable vector based on a sequence tree [HP05].

A node of the tree is annotated with the sum of the annotations of its children.




(a) (b) (c)

Figure 3.16: **Associativity property of a sequence tree.** Sequence trees with different topologies maintain the property that a node is annotated with the sum of the annotations of its children.

A leaf of the tree is annotated with a value of one.

Figure 3.15 illustrates the sequence tree. The ordinal numbers shown in square brackets are for illustration purposes only and are not part of the data structure.

A vector corresponds to a mathematical expression in which the addition function is applied to a value of one. Addition is both associative and commutative, so the vector maintains the property that the annotation of a node contains the sum of the number of leaves in the subtree that it suspends, regardless of the topology of that subtree.

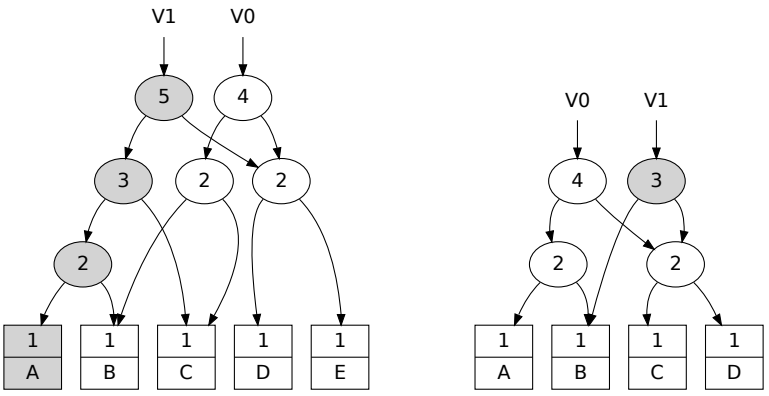
Figure 3.16 illustrates the associativity property

To locate a leaf with a given target ordinal the annotations of the children of the root node are examined. If the target ordinal is greater than or equal to the annotation of the left child then the right child is on the path. If the right path is chosen then the annotation of the left path is subtracted from the target ordinal number. If the left path is chosen then the target ordinal is unchanged. The comparison process continues at each node until a leaf is reached.

Figure 3.17 illustrates the insertion and removal of an element in an immutable sequence tree.

Figure 3.18 illustrates the growth of the immutable sequence tree.

The vector ADT can be restricted to implement an immutable array. To implement an immutable array a vector is populated with values before normal




(a) (b)

Figure 3.17: **Insertion and removal of an element in an immutable sequence tree.**
(a) Insertion of an element into an immutable sequence. Version V0 contains the sequence $\{[1]B, [2]C, [3]D, [4]E\}$. The operation $Insert([1]A)$ creates version V1 containing the sequence $\{[1]A, [2]B, [3]C, [4]D, [5]E\}$. The path created by the operation is shaded.
(b) Removal of an element from an immutable sequence. Version V0 contains the sequence $\{[1]A, [2]B, [3]C, [4]D\}$ The operation $Remove([1])$ creates version V1 containing the sequence $\{[1]B, [2]C, [3]D\}$. The path created by the operation is shaded.

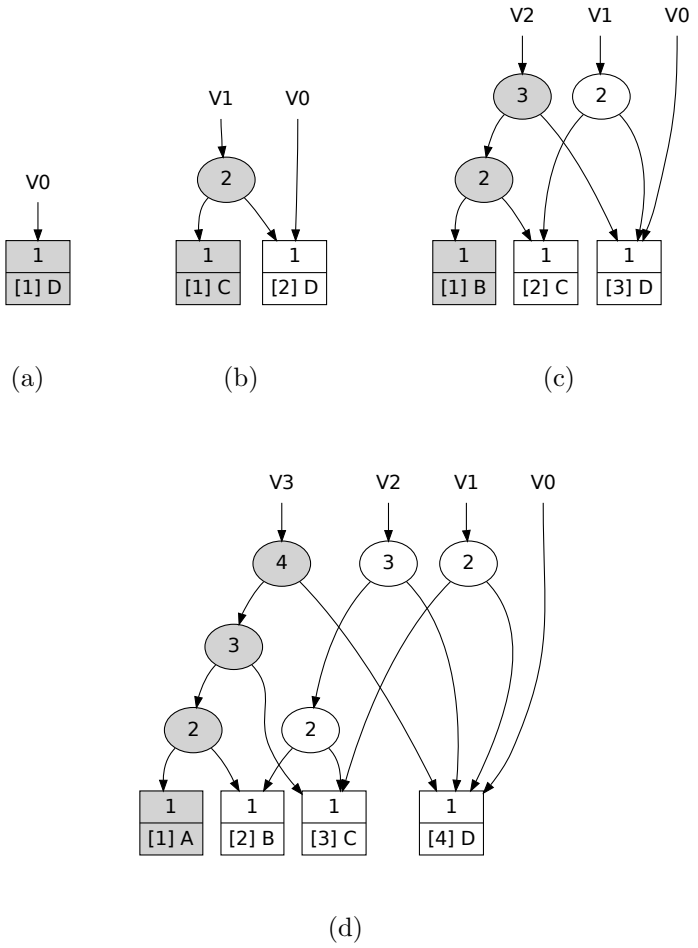



Figure 3.18: **Animation showing the growth of an immutable sequence tree** through a series of insertions. New versions of the data structure are created by each operation. In each case the path created by the operation is shaded. Version V3 represents the sequence $\{[1]A, [2]B, [3]C, [4]D\}$
(a) Initial data structure containing the value $D$.
(b) After $Insert([1]C)$
(c) After $Insert([1]B)$
(d) After $Insert([1]A)$

access is permitted. Replacement is a compound operation formed by an insert and a remove operation acting on elements with the same ordinal number. It is the only operation normally permitted by a vector implementing an array.

The sequence suffers from some of the same shortcomings as the priority queue, it requires the implementation of access functions which are specific to the vector ADT and it does not distinguish an empty vector from a non-existent vector.

### 3.5.8 Directed sequence

A new data structure, the directed sequence, addresses the shortcomings of the sequence tree. The comparison function of the sequence accesses only the node annotation. The directed sequence supports a sentinel and fully abstracts the ADT implementation from the functions of the Canonical Binary Tree.

The sequence requires that the annotations of the children of a node are examined to determine the path and this results in unnecessary accesses to nodes that are not on the path. To avoid these accesses a node should be annotated in such a way that the direction of the path can be determined without accessing the annotations of its children. This can be achieved by a regarding the annotation as a pair.

The second annotation, of the directed sequence, is the sum of the second annotations of the left and right children. The first annotation is set to the value of the second annotation of the left child. The first annotation is used to determine the path.

Figure 3.19 illustrates an example of a directed sequence.

To locate a leaf with a given target ordinal that ordinal is compared with the first annotation of the root node. If it is greater the right child is chosen otherwise the left child is chosen. This process is repeated until a leaf is reached. When a right child is chosen the second annotation of the child is subtracted from the target ordinal.

The sentinel is always the right-most leaf of the Canonical Binary Tree. To support the $Back()$ and $Push\_back()$ functions the sentinel and the leaf to the left of the sentinel must be annotated in such a way that they can be found without specifying an ordinal. The ADT specifies a special value which causes the second annotation of the root to be used as the access argument. The second value of the root is the number of elements in the sequence, including the sentinel.

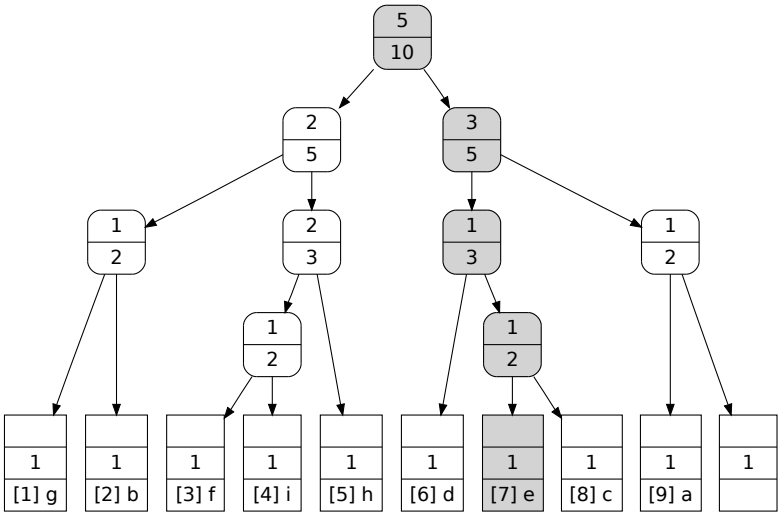



Figure 3.19: **Example Directed sequence** containing the values: $\{[1]g, [2]b, [3]f, [4]i, [5]h, [6]d, [7]e, [8]c, [9]a\}$. The shaded path illustrates the access to the leaf with an ordinal of seven. The sentinel is the right-most leaf. The first annotation is shown above the second annotation.

| Canonical Binary Tree specialisation | |
|---|---|
| $annotator(< a, b >, < c, d >)$ | $< b, b + d >$ |
| $identity$ | $<, 1 >$ |

| API function | Canonical Binary Tree access function |
|---|---|
| $Insert(ordinal)$ | $insert(ordinal)$ |
| $Remove(ordinal)$ | $delete(ordinal)$ |
| $Query(ordinal)$ | $query(ordinal)$ |
| $Push\_front()$ | $insert(0)$ |
| $Pop\_front()$ | $delete(0)$ |
| $Front()$ | $query(0)$ |
| $Push\_back()$ | $insert(\star)$ |
| $Pop\_back()$ | $delete(\star - 1)$ |
| $Back()$ | $query(\star - 1)$ |

Table 3.4: **Directed sequence implementation.** The Canonical Binary Tree can be specialised implement a directed sequence and its access functions can be adapted to present a vector ADT to the application. The second annotation of the root is represented by a star.

Table 3.4 contains all of the information required to specialise the Canonical Binary Tree so that it implements the vector ADT. The value of the second annotation of the root node is represented by a star.

The functions of the Canonical Binary Tree are complete so all values of the access parameter are valid arguments. An ordinal value less than or equal to one refers to ordinal number one. However, an ordinal number equal to or higher than the number of elements in the sequence refers to the sentinel. The $Remove()$ function verifies that its access argument is less than the second annotation of the root.

The directed sequence permits access to all leaves using their ordinal number. The ordinal is relative to the start of the sequence. Array indexes map to ordinals which start at one. For example, the $Query(0)$, $Query(1)$ and $Front()$ functions have the same effect.

An immutable array can be created by restricting the functions of the vector.

The sequence implementation requires that a value is retained and decremented while determining the path, so the function that determines the path through the data structure is specific to the vector ADT. This is unfortunate as some of the generality of the Canonical Binary Tree must be sacrificed to support the sequence. In our implementation the Canonical Binary Tree functions are supplied with an additional parameter which alters the mechanism for determining the path when implementing a sequence.

The sequence implementation requires that the ADT has access to the second annotation of root node. This is unfortunate as annotations are structural information that should not be exposed to the application.

### 3.5.9 Previous work

Anderson describes how the comparison operations used to determine the path through an interval tree can be confined to those nodes that are actually on the path [AN95]. We extend this idea and apply it to immutable min-trees, deques and sequence trees.

# Bibliography

[AN95] Arne Andersson and Stefan Nilsson. Efficient implementation of suffix trees. *Softw. Pract. Exper.*, 25:129–141, February 1995.

[HP05] R Hinze and R Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Prog.*, 16(02):197–217, 2005.

[Jos99] Nicolai M. Josuttis. *The C++ Standard Library: A tutorial and reference.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.