with the following mechanics

1. The reference x may be modified by swapping, assigning cell element values and zero-clearing cell element values, but must be restored to a 0-element sized array before the **delete** statement. Otherwise, the meaning of the statement is undefined.
2. If the reference x is the empty array upon the **delete** statement execution, the zero-cleared memory is reclaimed by the system.

With reversible, dynamic arrays of varying types and dimensionality, we must be extremely careful when updating and assigning values, to ensure we maintain reversibility and avoid irreversible statements. Therefore, when assigning or updating integer elements with one of the reversible assignment operators, we prohibit the cell value from being reference on the right hand side, meaning the following statement is prohibited

$$x[5] \mathrel{+}= x[5] + 1$$

However, we do allow other, initialized (non-zero-cleared array elements) to be referenced in the right hand side of the statement.

2.6 Local blocks

2.7 ROOPL++ Expressiveness

By introducing dynamic lifetime objects and by allowing objects to be referenced multiple times, we can express non-trivial programs in the reversible setting. To demonstrate the capacities, expressiveness and possibilities of ROOPL++ , the following section presents reversible data structures written in ROOPL++ .

2.7.1 Linked List

Haulund presented a Linked List implemented in ROOPL. The implementation featured a *ListBuilder* and a *Sum* class, required to determine and retain the sum of a constructed linked list as ROOPL's statically scoped object blocks would deallocate automatically after building the full list. In ROOPL++ we do not face the same challenges and the implementation becomes much more forward. Figure 2.3 implemented a *LinkedList* class, simply has the head of the list and the list of length as its internal fields. For demonstration, the class allows extension of the list by either appending or prepending cell elements to the list. In either case, we first check if the *head* field is initialized. If not, the cell we are either appending or prepending simply becomes the new head of the list. If we are appending a cell the Cell-class *append* method is called on the *head* cell with the new cell as its only argument. When prepending, the existing head is simply appended to the new cell and the new cell is set as head of the linked list.

Figure 2.4 shows the *Cell* class of the linked list which has a *next* and a *data* field, a constructor and the *append* method. The append method works by recursively looking through the linked cell nodes until we reach the end of the free list, where the *next* field has not been initialized yet.

When we find such a cell, we simply swap the contents of the *next* and *cell* variables, s.t. the cell becomes the new end of the linked list.

An interesting observation, is that the *append* method is called an additional time *after* setting the cell as the new end of the linked list. In a non-reversible programming language, we would simply call append in the else-branch of the first conditional. In the reversible setting, this is not an option, as the append call would modify the value of the *next* and *cell* variables and as such, corrupt the control flow as the exit condition would be true after executing both the then- and else-branch of the conditional. This "wasted" additional call with a **nil** value *cell* is a recurring technique in the following presented reversible data structure implementations.

```
class LinkedList
    Cell head
    int listLength

    method insertHead(Cell cell)
        if head = nil & cell != nil then
            head <=> cell              // Set cell as head of list if list is empty
        else skip
        fi head != nil & cell = nil

    method appendCell(Cell cell)
        call insertHead(cell)          // Insert as head if empty list

        if head != nil then
            call head::append(cell)    // Iterate until we hit end of list
        else skip
        fi head != nil

        listLength += 1                // Increment length

    method prependCell(Cell cell)
        call insertHead(cell)          // Insert as head if empty list

        if cell != nil & head != nil then
            call cell::append(head)    // Set cell.next = head. head = nil after execution
        else skip
        fi cell != nil & head = nil

        if cell != nil & head = nil then
            cell <=> head              // Set head = cell. Cell is nil after execution
        else skip
        fi cell = nil & head != nil

        listLength += 1                // Increment length

    method length(int result)
        result ^= listLength
```

Figure 2.3: Linked List class

2.7.2 Binary Tree

Figures 2.5, 2.6 and 2.7 shows the implementation of a binary tree in form of a rooted, unbalanced, min-heap. The *Tree* class shown in figure 2.5 has a single root node field and the three methods *insertNode, sum* and *mirror*. For insertion, the *insertNode* method is called from the *root*, if it is initialized and if not, the passed node parameter is simply set as the new root of the tree. The

```
class Cell
    Cell next
    int data

    method constructor(int value)
        data ^= value

    method append(Cell cell)
        if next = nil & cell != nil then
            next <=> cell           // Store as next cell if current cell is end of list
        else skip
        fi next != nil & cell = nil

        if next != nil then
            call next::append(cell) // Recursively search until we reach end of list
        else skip
        fi next != nil
```

Figure 2.4: Linked List cell class

insertNode method implemented in the *Node* class shown in figure 2.6 first determines if we need to insert left or right but checking the passed value against the value of the current node. This is done recursively, until an uninitialized node in the correct subtree has been found. Note that the value of node we wish to insert must be passed separately in the method call as we otherwise cannot zero-clear it after swapping the node we are inserting with either the right or left child of the current cell.

Summing and mirroring the tree works in a similar fashion by recursively iterating each node of the tree. For summing we simply add the value of the node to the sum and for mirroring we swap the children of the node and then recursively swap the children of the left and right node, if initialized. The sum and mirror methods are implemented in figure 2.7.

2.7.3 Doubly Linked List

Finally, we present the reversible doubly linked list, shown in figures 2.8-2.10. A *cell* in a doubly linked list contains a reference to itself named *self*, a reference to its left and right neighbours, a data and an index field. As with the linked list and binary tree implementation the *DoubleLinkedList* class has a a field referencing the head of the list and its *appendCell* method is identical to the one of the linked list.

When we append a cell to the list, we first search recursively through the list until we are at the end. The new cell is then set as *right* of the current cell. A reference to the current self is created using the **copy** statement, and set as *left* of the new end of the list, thus resulting on the new cell being linked to list and now acting as end of the list.

```
class Tree
    Node root

    method insertNode(Node node, int value)
        if root = nil & node != nil then
            root <=> node
        else skip
        fi root != nil & node = nil

        if root != nil then
            call root::insertNode(node, value)
        else skip
        fi root != nil

    method sum(int result)
        if root != nil then
            call root::getSum(result)
        else skip
        fi root != nil

    method mirror()
        if root != nil then
            call root::mirror()
        else skip
        fi root != nil
```

Figure 2.5: Binary Tree class

```
class Node
    Node left
    Node right
    int value

    method setValue(int newValue)
        value ^= newValue

    method insertNode(Node node, int nodeValue)
        // Determine if we insert left or right
        if nodeValue < value then
            if left = nil & node != nil then
                // If open left node, store here
                left <=> node
            else skip
            fi left != nil & node = nil

            if left != nil then
                // If current node has left, continue iterating
                call left::insertNode(node, nodeValue)
            else skip
            fi left != nil
        else
            if right = nil & node != nil then
                // If open right node spot, store here
                right <=> node
            else skip
            fi right != nil & node = nil

            if right != nil then
                // If current node has, continue searching
                call right::insertNode(node, nodeValue)
            else skip
            fi right != nil
        fi nodeValue < value
```

Figure 2.6: Binary Tree node class

```
method getSum(int result)
    result += value                  // Add the value of this node to the sum

    if left != nil then
        call left::getSum(result)   // If we have a left child, follow that path
    else skip                       // Else, skip
    fi left != nil

    if right != nil then
        call right::getSum(result)  // If we have a right child, follow that path
    else skip                       // Else, skip
     fi right != nil

method mirror()
    left <=> right                  // Swap left and right children

    if left = nil then skip
    else call left::mirror()        // Recursively swap children if left != nil
    fi left = nil

    if right = nil then skip
    else call right::mirror()       // Recursively swap children if right != nil
    fi right = nil
```

Figure 2.7: Binary Tree node class (cont)

```
class DoublyLinkedList
    Cell head
    int length

    method appendCell(Cell cell)
        if head = nil & cell != nil then
            head <=> cell
        else skip
        fi head != nil & cell = nil

        if head != nil then
            call head::append(cell)
        else skip
        fi head != nil

        length += 1
```

Figure 2.8: Doubly Linked List class

```
class Cell
int data
int index
Cell left
Cell right
Cell self

method setData(int value)
    data ^= value

method setIndex(int i)
    index ^= i

method setLeft(Cell cell)
    left <=> cell

method setRight(Cell cell)
    right <=> cell

method setSelf(Cell cell)
    self <=> cell
```

Figure 2.9: Doubly Linked List Cell class

```
    method append(Cell cell)
        if right = nil & cell != nil then   // If current cell does not have a right neighbour
            right <=> cell                  // Set new cell as right neighbour of current cell

            local Cell selfCopy = nil
            copy Cell self selfCopy         // Copy reference to current cell
            call right::setLeft(selfCopy)   // Set current as left of  right neighbour
            delocal Cell selfCopy = nil

            local int cellIndex = index + 1
            call right::setIndex(cellIndex) // Set index in right neighbour of current
            delocal int cellIndex = index + 1
        else skip
        fi right != nil & cell = nil

        if right != nil then
            call right::append(cell)        // Keep searching for empty right neighbour
        else skip
        fi right != nil
```

Figure 2.10: Doubly Linked List Cell class (cont)