CS 225 – Data Structures

ZJUI – Spring 2021

Lecture 2: Sequence Data Structures

Klaus-Dieter Schewe

ZJU–UIUC Institute, Zhejiang University

International Campus, Haining, UIUC Building, B404

email: kd.schewe@intl.zju.edu.cn

2 Sequence Data Structures

We will look at the comparably simple data data structures of sequences building upon the example of the ADT $LIST(T)$

Sequences are data structures for finite sets, i.e. we have a function $s : \{1, \ldots, n\} \rightarrow T$ for an arbitrary set T, for which the total order $\leq$ on $\{1, \ldots, n\}$ is relevant

We will distinguish between **lists** as already sketched, **stacks**, and **queues**

For the latter two the access and other operations are restricted, which makes them easier to handle

Furthermore, for lists we will investigate to have direct access to elements or not

In the next part of this course we will address data structures for finite sets and multisets, for which such an order is not relevant

2.1 Lists with Direct Access

Computer (main) memory is linearly organised with a fixed **address** for each **word** in memory

A **word** is a bit string of length n (common standard is now $n = 64$)

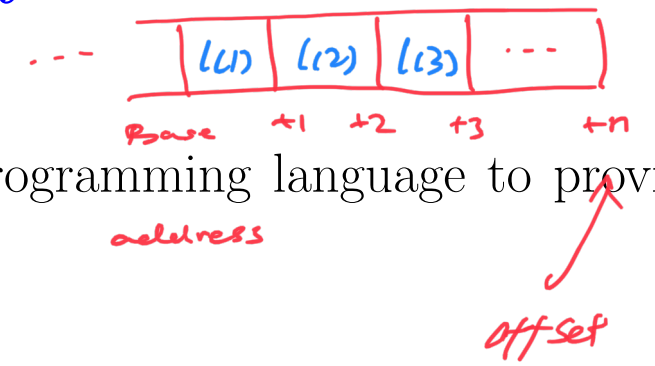
Without loss of generality we may assume that addresses are also given by bitstrings of word-length—if this is insufficient, then there are techniques to address even larger memory (refer to a course on assembler)

Direct access to a location in memory means to be able to provide a unique symbolic name for this location

can be mapped to some address

That is, for lists with direct access to their elements we require a way to have symbolic names for all locations $list(i)$, which defines the address of the i'th element of the list in memory

Arrays



The common way in an assembler or a programming language to provide direct access is by using **variables**

Variables (usually after compiling, binding and loading) provide unique addresses for memory locations

An **array** A provides a name for a contiguous sequence of $n \cdot c$ words

Here n is the length of the array, and c is the space (in number of words) reserved for storing a value at the address provided by $A[i]$ for $0 \leq i \leq n-1$ (usually $c = 1$)

That is, the notation $A[i]$ uses a **base address** associated with the name A of the array and an **offset** i

Using Arrays to Represent Lists

It is therefore a natural idea to use arrays to represent lists, provided the length of the list is at most the size of the array

In order to keep track of the length use an additional variable *length*

Thus, if a list ℓ is represented by an array A and a variable *length*, some operations become fairly simple:

unbounded arrays to represent list

- $length(\ell)$ just requires to access *length*, which requires time constant time, i.e complexity in $\Theta(1)$

- $get(\ell, i)$ requires to execute

  IF $i \leq length$ **THEN** $get(\ell, i) := A[i-1]$ **ELSE** $get(\ell, i) := undef$ **ENDIF**

  This is also in $\Theta(1)$

  ↓ constant

Time Complexity of List Operations / 1

- Analogously, $set(\ell, i, x)$ is realised as

$$\textbf{IF } i \leq length \textbf{ THEN } A[i-1] := x \textbf{ ENDIF} ,$$

  which again has time complexity in $\Theta(1)$

- The $append(\ell, x)$ operation requires to execute (with the length n of the representing array A)

```
IF      length < n
THEN  PAR
            length := length + 1
            A[length] := x
      ENDPAR
ENDIF
```

  with time complexity in $\Theta(1)$—we will later address the case $length = n$

Time Complexity of List Operations / 2

For membership checking the direct access has hardly any advantage: $in(x, \ell)$ requires the execution of

SEQ $in(x, \ell) :=$ **false**
FORALL i **WITH** $1 \leq i \leq length$
DO
IF $A[i-1] = x$ **THEN** $in(x, \ell) :=$ **true ENDIF**
ENDDO
ENDSEQ

As this requires the cecking of all list elements, the time complexity is in $\Theta(m)$, where m is the length of the list

The rule above can be optimised by sequential search, but without impact on the complexity

Time Complexity of List Operations / 3

For a sublist test $sublist(\ell_1, \ell_2)$ we have to iterate over the elements of ℓ_1

That is, initialise $i := 0$ and $j := 0$ and iterate the rule

IF $j = length_2$
THEN $sublist(\ell_1, \ell_2) :=$ **false**
ELSE **IF** $i < length_1$
THEN **IF** $A[i] = B[j]$
THEN **PAR** $i := i + 1$ $j := j + 1$ **ENDPAR**
ELSE $j := j + 1$
ENDIF
ELSE $sublist(\ell_1, \ell_2) :=$ **true**
ENDIF
ENDIF

Clearly, the time complexity is in $\Theta(m)$, where m is the length of ℓ_1

Time Complexity of List Operations / 4

- For equality checking in $\Theta(m)$ we may use the following rule:

$$
\begin{array}{ll}
\textbf{IF} & length_1 = length_2 \wedge \forall i.0 \leq i \leq length_1 - 1 \rightarrow A[i] = B[i] \\
\textbf{THEN} & eq(\ell_1, \ell_2) := \textbf{true} \\
\textbf{ELSE} & eq(\ell_1, \ell_2) := \textbf{false} \\
\textbf{ENDIF} &
\end{array}
$$

- The operation $delete(\ell, i)$ can be realised in time $O(m)$:

$$
\begin{array}{ll}
\textbf{IF} & length \neq 0 \\
\textbf{THEN} & \textbf{PAR} \\
 & \quad length := length - 1 \\
 & \quad \textbf{FORALL } j \textbf{ WITH } i - 1 \leq j \leq length - 2 \\
 & \quad\quad \textbf{DO } A[j] := A[j+1] \textbf{ ENDDO} \\
 & \textbf{ENDPAR} \\
\textbf{ENDIF} &
\end{array}
$$

Time Complexity of List Operations / 5

- The concatenation operation $concat(\ell_1, \ell_2)$ is similar to the $append()$ operation using the following rule:

```
IF      length_1 + length_2 < n
THEN  PAR
            length_1 := length_1 + length_2
            FORALL i WITH 0 ≤ i ≤ length_2 − 1
                  DO   PAR
                              A[length_1 + i] := B[i]
                       ENDPAR
                  ENDDO
      ENDPAR
ENDIF
```

with time complexity in $\Theta(m)$, where m is the length of ℓ_2—we will later address the case $length_1 + length_2 = n$

Time Complexity of List Operations / 6

The insertion operation $insert(\ell, i, x)$ requires shifting elements in the array, which again can be done in $O(m)$ time:

```
IF      length < n
THEN  PAR
            length := length + 1
            FORALL j WITH i ≤ j ≤ length
              DO A[j] := A[j − 1] ENDDO
            A[i − 1] := x
      ENDPAR
ENDIF
```

We will discuss the possibilities for the operation $sort(\ell)$ separately

First we will explore the handling of operations $insert()$, $concat()$ and $append()$ for the case, where the length n of the array is insufficient

Space Allocation

Operations *insert*(), *concat*() and *append*() as defined above extend the length of a list

If the length exceeds the length of the representing array the operations above are undefined

if not full (must have space)

In order to extend the specification we need to dynamically allocate space, in other words: define a new, larger array to represent the extended list

The idea is easily realised by a function *allocate*, which

if not enough, just double the size of the old one, copy.

- defines a new array with double the length of the given one
- copies the elements from the given array to the new one

Then the extensions of *insert*(), *concat*() and *append*() first execute *allocate*, then proceed as before

Amortised Complexity – Idea

分期偿还

The question is how space allocation affects the time complexity of the list operations

Clearly, *allocate* requires time in $O(n) = O(m)$

Consider first the case, where we have only *append*() operations changing the list's length

Then we can image that each time *append*() is executed without space allocation we count the required time three times, which still remains in $O(1)$, but consider the additionally counted time as an "investment"

Then after n such executions (starting from an empty list) the "accumulated investment" is just the time required for the space allocation and copying, i.e. the "investment has been amortised"

In other words: every single *append*() operation contributes a share to the necessary space allocation such that on average every *append*() still requires time in $O(1)$

Normal and Exceptional Case

Let us look at amortised complexity for list data structures in more detail

In general, for the ADT operations of interest we have a **normal case** and an **exceptional case** that occurs from time to time

For lists with direct access the operations of interest are *append*(), *concat*(), *insert*() and *delete*()—for the other operations discussed above there is no exceptional case

In the exceptional case another operation must be executed first

- For *append*(), *concat*() and *insert*() the exceptional case arises, when the operation will exceed the length of the representing array, in which case the execution of an *allocate* operation is required

- For *delete*() the exceptional case arises, when the operation will decrease the required fraction of the representing array below 1/4 of the available size; in this case the execution of a *deallocate* operation is required

Amortised Complexity

- *allocate* creates a new array of doubled size and copies all elements of the previous list to the new array

  If the complexity of a single copying operation is $c \in O(1)$, then *allocate* requires time complexity $c \cdot n \in O(n)$, where n is the length of the given array

- *deallocate* creates a new array of half the size and copies all elements of the previous list to the new array

  Then the complexity is $c \cdot n/2 \in O(n)$, where the length of the given array is $2n$

The complexity count for the *allocate* and *deallocate* is then distributed to the triggering operations

Their complexity becomes **amortised complexity** this way

Contribution of Operations to Exceptional Cases

Each of the operations (regardless in which case) makes a contribution to later *allocate*/*deallocate* operations

This contribution is $2c$ for the case of *append*() and *insert*(), c for *delete*(), and $2c \cdot |\ell|$ for *concat*(), where ℓ is the list to be concatenated at the end of the given list

Let us assume a counter C capturing the accumulated contributions by the operations of interest and the consumption by *allocate* and *deallocate*

- The initial value of C is 0
- Whenever an operation of interest is executed, its contribution is added to C
- Whenever *allocate* or *deallocate* are executed, their needed complexity count is subtracted from C

Feasibility of Amortisation

Proposition. If the initial length of the representing array is n_0, then after each execution of an *allocate* or *deallocate* operation the counter C will have a value $\geq c \cdot n_0$. In particular, we always have $C \geq 0$.

This shows that the contributions of the operations of interest are sufficient to cover the complexity count for all necessary *allocate* and *deallocate* operations

We therefore get the following amortised complexity:

- The amortised time complexity of $concat(\ell_1, \ell_2)$, $insert(\ell, i, x)$ or $delete(\ell, i)$ is in $O(n)$, where n is the length of the representing array

- The amortised time complexity of $append(\ell, x)$ is in $\Theta(1)$

Proof of the Proposition

Without loss of generality we can ignore the *concat*() operation, as its complexity for the normal case and its contribution to the counter C are the same as m *append*() operations

We proceed by induction over the length of the sequence of *allocate* and *deallocate* operations

The first such operation can only be *allocate*, and it can only be executed after at least n_0 *insert*() or *append*() operations have been executed

Each such operation contributed $2 \cdot c$ to the counter C, so when *allocate* is executed, we have $C \geq 2 \cdot n_0 \cdot c$, which is reduced by $n_0 \cdot c$ due to the n_0 necessary copy operations

For the induction step we consider two cases

Case 1. Assume that the length of the representing array is $n > n_0$ and an *allocate* operation becomes necessary

Proof / cont.

Then there must have been at least $n/2$ *insert*() or *append*() operations since the previous *allocate* or *deallocate*, i.e. at least $n \cdot c$ has been added to the counter C

The *allocate* operation creates a new array of length $2n$ and executes n copy operations thereby reducing the counter C by $n \cdot c$ $\frac{n}{2} \cdot 2c$

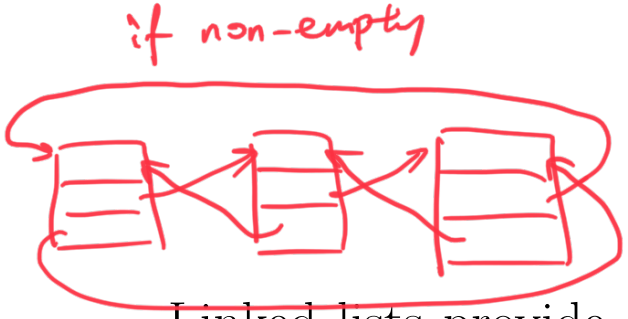
By induction we had $C \geq n_0 \cdot c$ after the previous *allocate* or *deallocate*, hence we get again $C \geq n_0 \cdot c$

Case 2. Assume that the length of the representing array is $2n > n_0$ and a *deallocate* operation becomes necessary

This is only possible, if there have been $2n/4$ more *delete*() operations than *insert*() and *append*() operations, and thus C has been increased at least by $n \cdot c/2$

Then the *deallocate* operation creates a new array of length n and executes $n/2$ copy operations thereby reducing the counter C by $n \cdot c/2$

By induction we had $C \geq n_0 \cdot c$ after the previous *allocate* or *deallocate*, hence we get again $C \geq n_0 \cdot c$, which completes the proof



2.2 Linked Lists

Linked lists provide an alternative way to represent lists without direct access to each list element

Instead each element of a list is represented by a pair or triple containing also a pointer/reference to the next list element and/or the next and the previous element—we refer to each such pair or triple as a **node**

We therefore distinguish **Singly Linked Lists** and **Doubly Linked Lists**

In addition, in both cases we use a **dummy node** with forward reference to the first list element and (in case of doubly linked lists) backward reference to the node containing the last element of the list

The node with the last list element contains a forward reference to the dummy node

In this way the empty list comprises only the dummy node, which references itself (forward and backward)

Abstract Representation

Let us concentrate on doubly linked lists, which we can represent a set of nodes, where each node is a quadruple $(id, item, forward, backward)$ comprising

- an identifier id of the node,
- an $item$, which is an element of a set T or the specal value $\perp$ for the dummy node,
- an identifier $forward$ for the next node in the list, and
- an identifier $backward$ for the previous node in the list

undefined

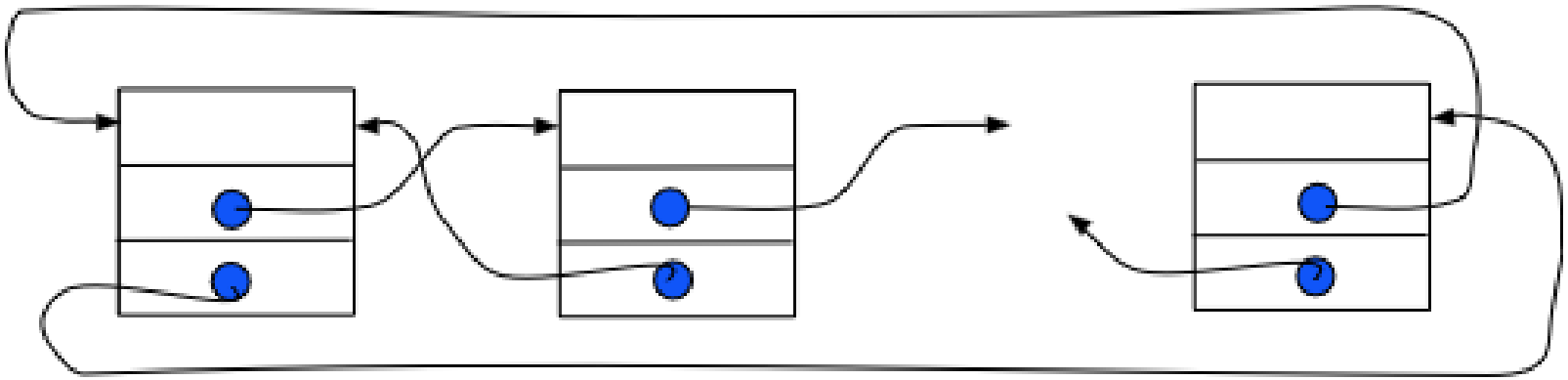
In addition, we have a variable $length$ taking as value the number of elements in the list

Only the identifier of the dummy node is known; all other identifiers are system-defined

Illustration of Linked Lists

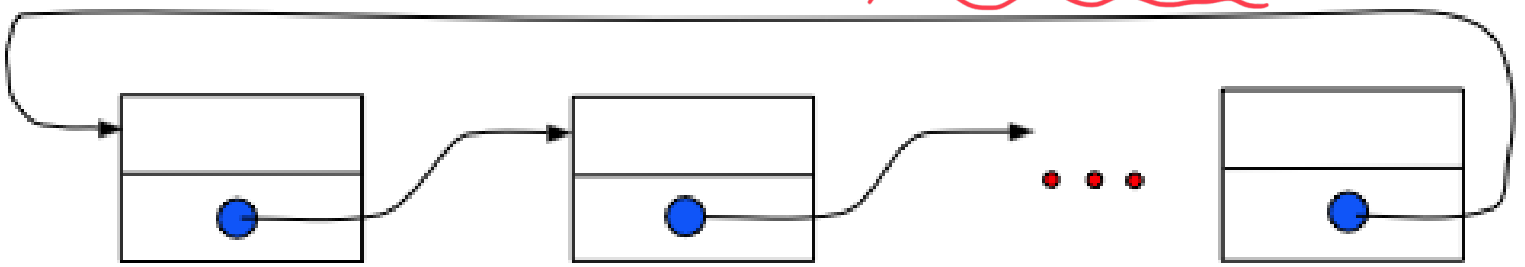
Doubly linked list:

back and forth.



Singly linked list:

only one direction



Complexity Analysis: Length and Get Operation

Clearly, $length(\ell)$ has complexity $\Theta(1)$, as it comprises just to access the variable $length$

For the $get(\ell, i)$ operation initialise $counter := 0$ and $address := id_0$, where id_0 is the identifier of the dummy node, i.e. $(id_0, \bot, f, b) \in \ell$ for some f and b

Then for $get(\ell, i)$ iterate the following rule (ignore the error handling for the case that i is out of range)

IF $counter \leq i$
THEN **LET** $f = \mathbf{I}f'.\exists x', b'.(address, x', f', b') \in \ell$
IN **PAR**
$address := f$
$counter := counter + 1$
ENDPAR
ELSE $result := \mathbf{I}x.\exists f', b'.(address, x, f', b') \in \ell$
ENDIF

That is, the algorithm follows the chain of forward references i-times, and then retrieves the value stored in the i'th node—thus, the time complexity is in $O(|\ell|)$

Complexity Analysis: Set Operation

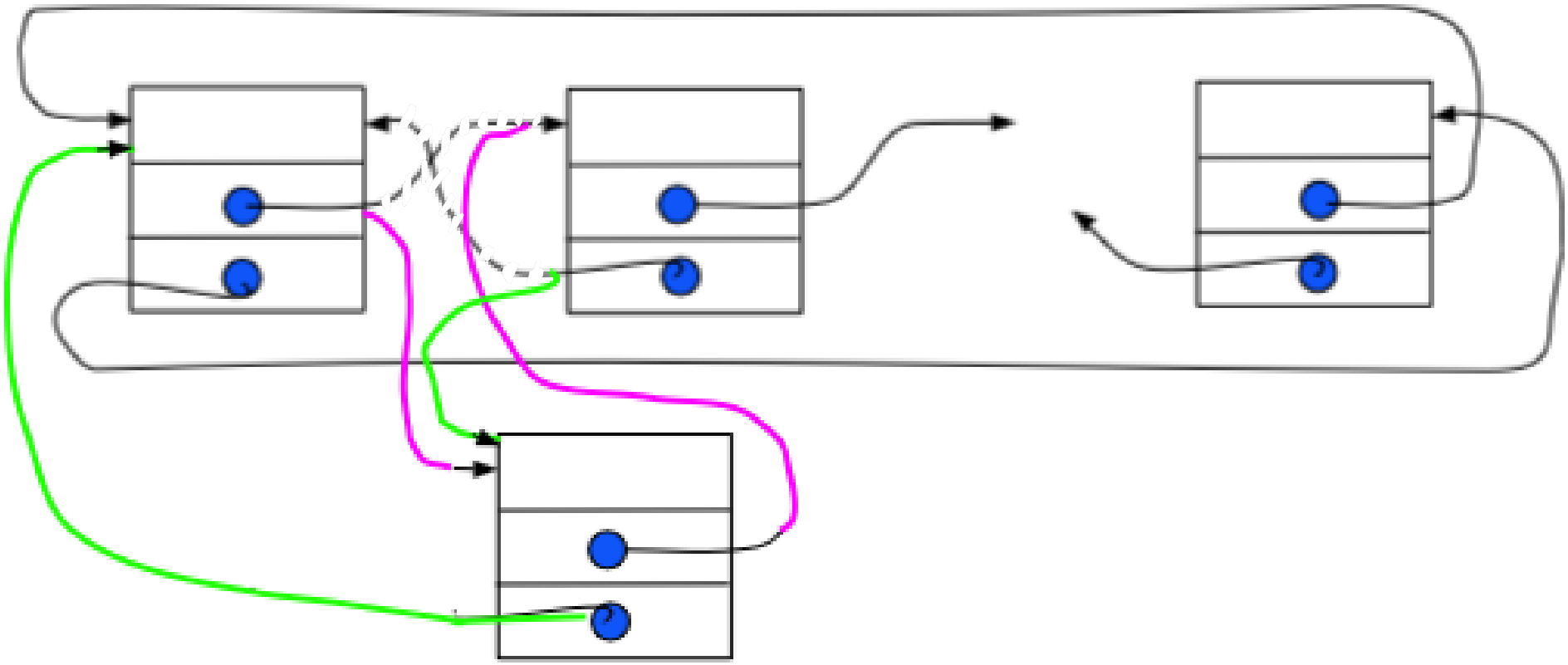
We proceed analogously intialising $counter := 0$ and $address := id_0$, where id_0 is the identifier of the dummy node, i.e. $(id_0, \bot, f, b) \in \ell$ for some f and b

Then $set(\ell, i, x)$ iterates the following rule (ignore again the error handling for the case that i is out of range)

IF $counter \leq i$
THEN **LET** $f = \mathbf{I}f'.\exists x', b'.(address, x', f', b') \in \ell$
IN **PAR**
$address := f$
$counter := counter + 1$
ENDPAR
ELSE **LET** $(address, x', f', b') \in \ell$
IN $\ell := \ell - \{(address, x', f', b')\} \cup \{(address, x, f', b')\}$
ENDLET
ENDIF

That is, the algorithm follows the chain of forward references i-times, and then updates the value stored in the i'th node—thus, the time complexity is in $O(|\ell|)$

Illustration



Complexity Analysis: Insert Operation

For *insert* we proceed analogously, i.e. follow the chain of references to the i'th node of the list

We have to add a new node and redirect the forward and backward references:

IF $counter \leq i$
THEN **LET** $f = \mathbf{I}f'.\exists x', b'.(address, x', f', b') \in \ell$
IN **PAR**
$address := f$
$counter := counter + 1$
ENDPAR
ELSE **LET** $(address, x', f', b') \in \ell \wedge (f', x'', f'', address) \in \ell$
IN **LET** $f := NewId$
IN $\ell := \ell - \{(address, x', f', b'), (f', x'', f'', address)\}$
$\cup\{(address, x, f, b'), (f, x', f', address), (f', x'', f'', f)\}$
ENDLET
ENDIF

linear

Consequently, the time complexity is again in $O(|\ell|)$

Complexity Analysis: Append and Concatenation

The most significant difference to the representation of lists by "unbounded" arrays and doubly-linked lists comes with the *append* and *concat* operations

For concatenation we only have to link the last node of the first list to the first node of the second list, and the last node of the second list to the dummy node of the first list

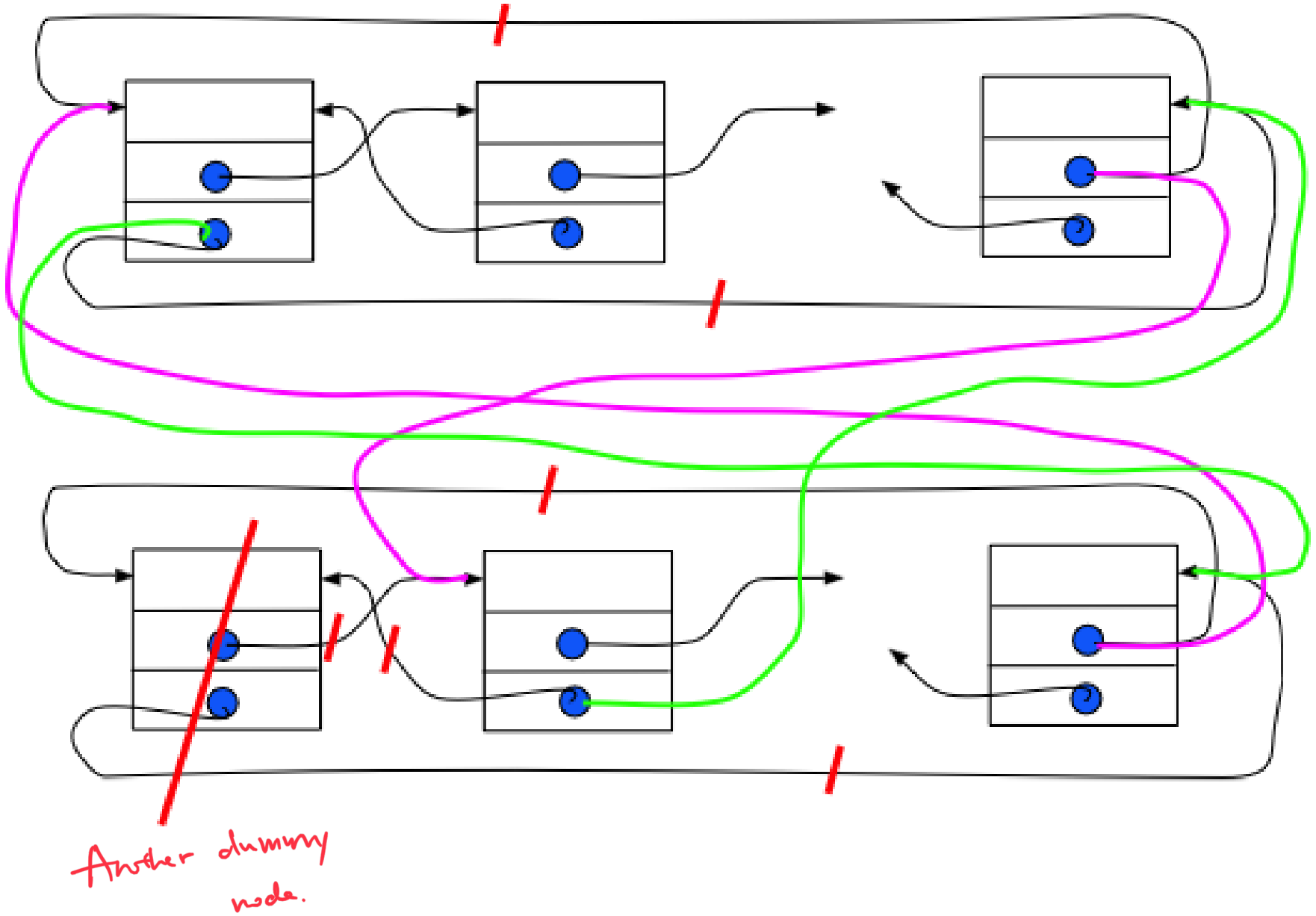
This is done by the following rule:

$$
\begin{array}{ll}
\textbf{LET} & (id_0, \bot, f_0, b_0) \in \ell_1,\ (b_0, z, id_0, b_1) \in \ell_1, \\
 & (id'_0, \bot, f'_0, b'_0) \in \ell_2,\ (b'_0, z', id'_0, b'_1 \in \ell_2,\ (f'_0, x, f'_1, b') \in \ell_2 \\
\textbf{IN} & \ell_1 := (\ell_1 - \{(id_0, \bot, f_0, b_0), (b_0, z, id_0, b_1)\}) \\
 & \quad \cup (\ell_2 - \{(id'_0, \bot, f'_0, b'_0), (b'_0, z', id'_0, b'_1), (f'_0, x, f'_1, b')\}) \\
 & \quad \cup \{(id_0, \bot, f_0, b'_0), (b_0, z, f'_0, b_1)\} \\
 & \quad \cup \{(b'_0, z', id_0, b'_1), (f'_0, x, f'_1, b_0)\}
\end{array}
$$

As only a constant number of nodes is updated, we have time complexity in $\Theta(1)$

append is done analogously to *concat* with time complexity in $\Theta(1)$

Illustration

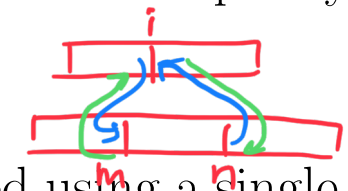


Complexity Analysis: Other Operations

For *delete* we proceed analogously to the *insert* operation

Note that in these two cases there is no need to allocate or deallocate space

The same applies for membership *in*, *sublist* and equality *eq*, i.e. in all cases we obtain complexity in $O(|\ell|)$



All operations we discussed can be simplified using a single ***splicing operation***, which cuts out a sublist between nodes with identifiers id_a and id_b, and (optinally) inserts this sublist into another list after the node with identifier id_c—the time complexity is in $O(1)$

For *insert* and *delete* the identifiers have to be determined by search—we leave details of splicing as an exercise

Sorting will be handled separately

Singly Linked Lists

The handling of singly linked lists is analogous: instead of nodes $(id, item, forward, backward)$ we only have nodes of the form $(id, item, forward)$

This only affects *append* and *concat* as the only operations exploiting the backward references (if we disregard optimisations for the search in *get*, *set*, *delete* and *insert*)

In these cases the address of the last node of the first list has to be obtained by following the chain of forward references, so the complexity will be in $\Theta(|\ell_1|)$

The remark concerning splicing remains the same

We leave details of singly linked lists as an exercise

Note that adding a single reference to the last node would preserve the benefits of doubly linked lists without the need of having backward references for all nodes

Advantages / Disadvantages

We have seen that the representation of lists as unbounded arrays has advantages for insertion and retrieval at a known position, whereas linked lists offer advantages for append and concatenation

Furthermore, a representation by an unbounded array always requires a contiguous space of memory, i.e. it is not applicable for huge sequences, in particular not, if secondary storage is required to store the whole list

Therefore, which list representation is to be applied depends on the application

It is of course possible to integrate both representations for list using doubly/singly linked lists to connect unbounded arrays