Linked Lists
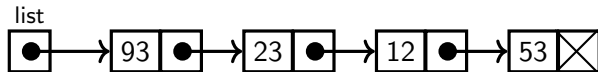
Linked Lists in Mem[−]

Linked list representing a list ⟨93, 23, 12, 53⟩:

Inserting at the beginning of a list:

```
1 void insert_beg (int list , int number) {
2   final newblock = allocate_memory (2);
3   Mem[ newblock ] = number ;
4   Mem[ newblock +1] = Mem[ l i s t ];
5   Mem[ l i s t ] = newblock ;
6 }
```

Check that it works, even if `Mem[list] == END`.
What is the complexity of `insert_beg`?
Does it depend on the size of the list?

`final list = 5071`
and

i	Mem[i]
0324	23
0325	3111
⋮	⋮
3111	12
3112	7479
⋮	⋮
3823	93
3824	0324
⋮	⋮
5071	3823
⋮	⋮
7479	53

1

Similarly to what we had before, each ☐☐ is realised as a block of two consecutive locations in `Mem`. The first location of such block stores a number and the second location stores the address of the following block.

The constant `list` is an address pointing to the first node, called the head pointer.

`END` indicates the end of the list (graphically as ⊠). Its value can be anything that is not a valid address, for example, `-1`. Most languages use `0` for this purpose, which, although theoretically is a valid address, is never one in practice.

A linked list is empty whenever `Mem[list]` is equal to `END`.

An advantage of linked lists over arrays is that the length of linked lists is not fixed. We can insert and delete items as we want. On the other hand, accessing an entry on a specific position requires traversing the list.

To represent the list $\langle 93, 23, 12, 53 \rangle$ as a linked list do the following:

```
1  // allocating memory for the boxes
2  final list = allocate_memory(1);
3  final block0 = allocate_memory(2);
4  final block1 = allocate_memory(2);
5  final block2 = allocate_memory(2);
6  final block3 = allocate_memory(2);
7
8  // entries and links
9  Mem[list]=block0;
10 Mem[block0] = 93;
11 Mem[block0+1] = block1;
12 Mem[block1] = 23;
13 Mem[block1+1] = block2;
14 Mem[block2] = 12;
15 Mem[block2+1] = block3;
16 Mem[block3] = 53;
17 Mem[block3+1] = END;
```

Alternatively, use the function `insert_beg`:

```
1  // create empty list
2  final list = allocate_mem[1]
3  Mem[list] = END;
4
5  // insert entries
6  insert_beg(list,53);
7  insert_beg(list,12);
8  insert_beg(list,23);
9  insert_beg(list,93);
```

Deleting at the beginning

```
1 Boolean is_empty(int list) {
2   return (Mem[list] == END);
3 }
```

```
1 void delete_begin(int list) {
2   if is_empty(list) {
3     throw new EmptyListException("delete_begin");
4   }
5   final firstnode = Mem[list];
6   Mem[list] = Mem[firstnode+1];
7   freememory(firstnode,2);
8 }
```

Lookup

```
1  int value_at(int list, int index) {
2    int i = 0;
3    int cursor = list;
4    int nextnode = Mem[cursor];
5    while (true) {
6      if (nextnode == END) {
7        throw OutOfBoundsException;
8      }
9      if (i == index) {
10       break;
11     }
12     cursor = nextnode + 1;
13     nextnode = Mem[cursor];
14     i++;
15   }
16   return Mem[nextnode];
17 }
```

What is the time complexity of these operations?

(How does this compare to arrays?)

How would you implement `insert_end` and `delete_end` ?

Insert at the end

```
1  void insert_end(int list, int number) {
2    final newblock = allocate_memory(2);
3    Mem[newblock] = number;
4    Mem[newblock+1] = END;
5
6    int cursor = list;
7    while (Mem[cursor] != END){
8      cursor = Mem[cursor]+1;
9    }
10   Mem[cursor] = newblock;
11 }
```

The same by recursion

```
1  void insert_end (int list , int number) {
2    final startnode = Mem[list];
3    if (startnode == END) {
4      final newblock = allocate_memory (2);
5      Mem[newblock] = number;
6      Mem[newblock+1] = END;
7      Mem[list] = newblock;
8    } else {
9      insert_end (startnode+1,number);
10   }
11 }
```

Search is a procedure which finds the position (counting from 0) where a value, given as a parameter, is stored in the array or linked list.

By "cost as a function of the number of elements" we mean: how does the number of items we have to inspect or modify grow as the number of elements in the structure grows? *Constant* means that the number of operations does not depend on the number of elements. *Linear* means that if we increase the number of elements in the structure by a factor of n, then the cost of the operation in the worst case will be multiplied by n too.

We compare costs by comparing the number of elements of the list we have to inspect (in the worst case) in order to finish the operation.

In practise, we choose between using an array or linked list depending on both relative frequency of use of the operations and their relative costs.

Comparison

If we store a list of `n` elements as an array (without spare space) or a linked list, what costs will the basic operations of lists have as a function of the number of elements in the list (when the list is seen as an ADT)?

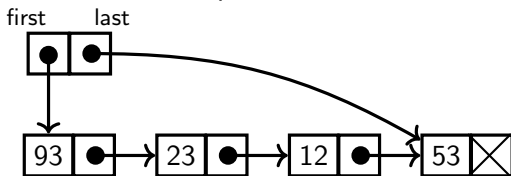|	Array	Linked List
access data by position		
search for an element		
insert an entry at the beginning		
insert an entry at the end		
insert an entry (on a certain position)		
delete first entry		
delete an entry		
concatenate two lists		

Comparison (solution)

If we store a list of `n` elements as an array (without spare space) or a linked list, what costs will the basic operations of lists have as a function of the number of elements in the list (when the list is seen as an ADT)?

	Array	Linked List
access data by position	constant	linear
search for an element	linear	linear
insert an entry at the beginning	linear	constant
insert an entry at the end	linear	linear*
insert an entry (on a certain position)	linear	linear
delete first entry	linear	constant
delete entry `i`	linear	linear
concatenate two lists	linear	linear*

The stars indicate that it could be improved to constant time if we modified the representation slightly. As we'll see very soon…
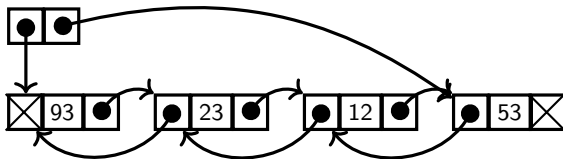
Modifications

Linked list with a pointer to the last node:

Fast `insert_end`
Slow `delete_end`

Doubly linked list:

Try yourself: Write `insert_beg`, `insert_end`, `delete_beg`
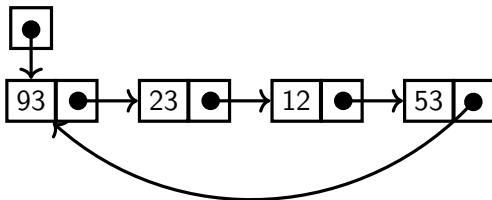and `delete_end` for those two representations.

If we remember where the first and the last block of a doubly linked list is stored, then inserting at the end and deleting from the end could be implemented in constant time.

As we play with code for manipulating singly and doubly linked lists, referring to memory locations via `MEM[a]`, `MEM[a+1]`, `MEM[a+2]` for the 3 elements of, for example, a doubly linked list node, can become somewhat tedious and error-prone. Therefore we will use the following shorthand:
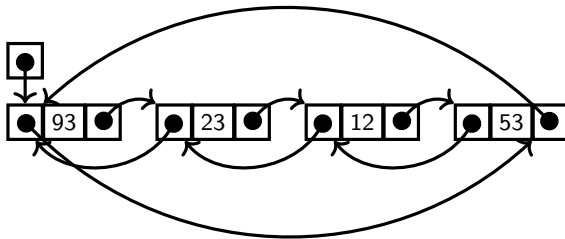
- For a singly linked list node we will use:

    - `a.val` for `MEM[a]`
    - `a.next` for `MEM[a+1]`

- For a doubly linked list node we will use:

    - `a.prev` for `MEM[a]`
    - `a.val` for `MEM[a+1]`
    - `a.next` for `MEM[a+2]`

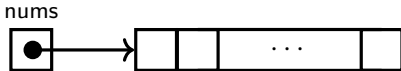**More Modifications**

Circular Singly
Linked List:

Circular Doubly
Linked List:

Try yourself: Write `insert_beg`, `insert_end`, `delete_beg`
and `delete_end` for those two representations.

Time for a quiz!

Let `nums` be the address of an array of integers of length `len`.

Which of the following algorithms creates a **linked list** faster?

```
1 final list = allocate_mem[1];
2 Mem[list] = END;
3 for (int i=0; i < len; i++) {
4   insert_end(list, nums.next);
5 }
```

```
1 final list = allocate_mem[1];
2 Mem[list] = END;
3 for (int i=len-1; i <= 0; i--) {
4   insert_beg(list, nums.next);
5 }
```

If we do not keep track of the last node then the second algorithm is faster. This is because every `insert_beg` takes constant time to execute whereas when running `insert_end` we need, every time we insert, to traverse the list to the end before actually adding the new element, and each time the list grows by one element so in total we have to traverse first a list of 0 element, then a list of 1 elements,..., until finally we traverse a list of `len` -1 elements. That is, in total we do $0+1+2+3+\ldots(\text{len} -1)$, traversal steps, that is:

$$0+1+2+3+\cdots+(\text{len} -1) = \frac{\text{len}\,(\text{len} -1)}{2}$$