2 Singly Linked Lists

2.1 Objectives

The learning objectives of this laboratory session are:

- To understand the challenges of implementing singly-linked lists and of the special cases of stacks and queues.
- To learn how to compare the various simple list models and choose the most suitable one for the given problem.
- To get hands-on experience with implementing singly-linked lists using dynamically allocated memory.
- To develop critical thinking concerning implementation decisions for (Singly-linked lists - SLLs).

The outcomes for this session are:

- The ability to compare simple list models and choose the most suitable one.
- Improved skills in C programming using dynamic memory allocation.
- A clear understanding of singly linked lists and the operations which they support.
- The ability to make implementation decisions concerning the singly-linked lists.
- The ability to decide when singly-linked lists are suitable in solving a problem.

2.2 Brief Theory Reminder

- A *list* is a finite ordered set of elements of a certain type.
- The elements of the list are called *cells* or *nodes*.
- A list can be stored
  - in a memory area of a size decided at compile time, or *statically*, using fixed-size arrays or, more often,
  - in a memory area of a size decided at run time, aka *dynamically* allocated.

With fixed size arrays, the lists cannot grow larger than the predefined size. One can use the implicit order of the elements of the one-dimensional array, or can use *cursors* to chain elements. When dynamically allocated memory is used as storage for SLLs, the order of nodes is set by *pointers*. Pointers may be either the C language pointers or indexes in a dynamically allocated array (in that case they are called *cursors*). As you know, dynamic allocation of memory uses the C program *heap*. If a list can be traversed in only one direction, from a current element to a next one, then it is said to be singly-linked. Doubly-linked lists can be traversed in both directions: from the current element to the one immediately succeeding it (the next element), or from a current element to another element which immediately precedes it (previous element). The structure of a node, in C, may be as in Listing 2.1, below:

Listing 2.1: Sample singly-linked list node type definition

```c
typedef struct node
{
  int key; /* an optional field identifying the data */
  /* other useful data fields */
  struct node *next; /* link to next node */
} NodeT;
```

A Singly-linked List Model

A singly-linked list model is illustrated in Figure 2.1. In the illustration, the list is referred to via a pointer $listPtr$. That pointer points to a header cell with three fields: a $count$ field, holding the current number of items in the list; a pointer, $first$, which points to the first cell in the list; and another pointer, $last$, pointing to the last item in the list.
All cells of the list are accessed via one these two pointer fields. Figure 2.1 shows a model where the list is circular – the last element is linked to the first.

A possible implementation may be as shown in Listing 2.2:

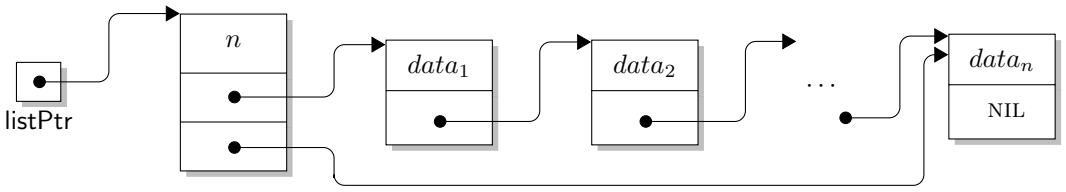

Figure 2.1: A singly-linked list model. In this model there is a header cell holding information about the list as a whole

Listing 2.2: Sample singly-linked list header cell type definition

```
typedef struct
{
  int count; /* an optional field */
  NodeT *first; /* link to the first node in the list */
  NodeT *last; /* link to the last node in the list */
} ListT;
```

2.3 Operations on a Singly-linked List

Creating a Singly-linked List

Consider the following steps which apply to *dynamic* lists:

1. Initially, the list is non-existent. This can be coded by setting the pointers $listPtr$ $NULL$.
2. Create an empty list, by creating its header cell, as shown in Listing 2.3.

Listing 2.3: Sample code to create an empty singly-linked list

```
/* Create an empty list */
ListT *createEmptySLL()
{
        ListT *listPtr = (ListT*)malloc(sizeof(ListT));
        if (listPtr)
        {
                listPtr->count = 0; // list empty
                listPtr->first = listPtr->last = NULL;
        }
        return listPtr;
}
```

3. The list holds nodes. Thus, it is a good choice to have a function (like the one in Listing 2.4) to create a SLL node and fill the data held in the node.

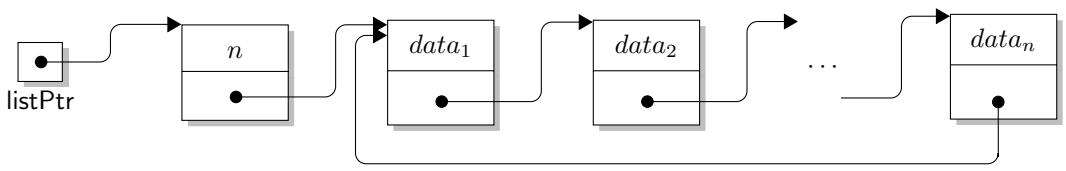

Figure 2.2: A circular singly-linked list model. In this model there is a header cell holding information about the list as a whole. Note that the last element is linked to the "first".

Listing 2.4: Sample code to create and fill a SLL node with data

```
/* Create a node and fill it with data */
NodeT *createSLLNode(int key)
{
    NodeT *p = (NodeT *)malloc(sizeof(NodeT));
    if (p)
    {
        // what is done here depends on the data stored at the node
        p->key = key; // assignment allowed as the key is of a primitive type
        p->next = NULL;
    }
    return p;
}
```

Inserting a Node in a Singly-linked List

The insert operation may have various forms. Thus, one may:

- **insertAtFront** – insert every new element at the beginning of the list. This is always used with stacks (the *push* operation of a stack) or queues (the *enqueue* operation).
- **insertAtRear** – insert every new element at the end of the list. Also known as an *append*. This is used, e.g. with queues (the *enqueue* operation of a queue).
- **insertInOrder** – every new element is inserted into a sorted list in ascending or descending order of its *key* field.

Listing 2.5 below shows the implementation for **insertAtFront** and **insertAtRear**.

Listing 2.5: Sample code for inserting a node at the beginning or end of a SLL

```
// Example code for insertAtFront; does not check for duplicate keys
int insertAtFront(ListT *listPtr, NodeT *p)
{
    if (listPtr)
    {
        // the list is non null
        p->next = listPtr->first;
        if (isEmpty(listPtr))
        {
            // p will be the sole node in the list
            listPtr->last = p;
        }
        listPtr->first = p;
        listPtr->count++;
        return 1; // success
    }
    return 0; // failure
}
// Example code for insertAtRear; does not check for duplicate keys
int insertAtRear(ListT *listPtr, NodeT *p)
{
        if (listPtr)
        {  // the list is non null
                if (isEmpty(listPtr))
                {  // p will be the sole node in the list
                        listPtr->first = p;
                }
                else
                { // non-empty list
                        listPtr->last->next = p;
                }
                listPtr->last = p;
                listPtr->count++; // increment number of nodes
                return 1; // success
        }
        return 0; // failure
}
```

- Insertion before a node given by its *key*. There are two steps to execute:
    1. Search for the node containing a *givenKey*, as shown by Listing 2.6.

Listing 2.6: Sample code for searching a node with a given key in a SLL, preceeding an insert operation

```
NodeT *q, *q1;
q1 = NULL; /* initialize */
q = listPtr->first;
while ( q != NULL )
{
    if ( q->key == givenKey ) break;
    q1 = q;
    q = q->next;
}
```

2. Insert the node pointed to by p, and adjust links, as Listing 2.7 shows.

Listing 2.7: Sample code for inserting a node with a given key in a SLL

```
if ( q != NULL )
{   /* node with key givenKey has address q */
    if ( q == listPtr->first )
    {   /* insert before the first node */
        p->next = listPtr->first; // chain with former first
        listPtr->first = p; // set p as first node
    }
    else
    {   /* inside the list */
        q1->next = p;
        p->next = q;
    }
    listPtr->count++; // increment number of nodes
    // success
}
// failure;
```

- Insertion after a node given by its *key*. Again, there are two steps to execute:

1. Search for the node containing the *givenKey*, as shown before in Listing 2.6.
2. Insert the node pointed to by p, and adjust links, as shown by Listing 2.8.

Listing 2.8: Sample code for inserting a node with a given key in a SLL

```
if ( q != NULL )
{
    p->next = q->next; /* node with key givenKey has address q */
    q->next = p;
    if ( q == listPtr->last ) listPtr->last = p;
    listPtr->count++; // increment number of nodes
    // success
}
    // failure
```

Finding a Node of a Singly-linked List

The nodes of a list may be accessed *sequentially*, gathering useful information as needed. Typically, a part of the information held at a node is used as a *key* for finding the desired information. The list is scanned linearly, and the node may or may not be present on the list. A function for searching a specific key may contain the sequence of statements presented in Listing 2.9.

Listing 2.9: Sample code for a function to find a node with a given key in a SLL

```
NodeT *find(ListT *listPtr, int givenKey)
{
    NodeT *p;
    p = listPtr->first;
    while ( p != NULL )
        if ( p->key == givenKey ) /* Note. This comparison does work for primitive types only
            */
        {
            return p; /* key found in cell p */
        }
        else
            p = p->next;
    return NULL; /* not found */
}
```

The node to be inserted may be created as shown at §2.3. We will assume here that the node to insert is pointed to by p.

Removing a Node of a Singly-linked List

When we are to remove a node from a list, there are some aspects to take into account: (i) list may be empty; (ii) list may contain a single node; (iii) list has more than one node. And, also, deletion of the first, the last or a node given by its key may be required. Thus we have the following cases:

1. Removing the first node of a list – illustrated by Listing 2.10.

   Listing 2.10: Sample code for deleting the first node in SLL

   ```
   NodeT *deleteFirst(ListT *listPtr)
   {
       NodeT *p;
       if ( listPtr->first != NULL )
       {
           // non-empty list
           p = listPtr->first;
           listPtr->first = listPtr->first->next;
           // free( p ); // free up memory
           listPtr->count--; // decrease the number of nodes
           if ( listPtr->first == NULL ) // list is now empty
               listPtr->last = NULL;
           return p;
       }
       return NULL;
   }
   ```

2. Removing the last node of a list – illustrated by Listing 2.11.

Listing 2.11: Sample code for deleting the last node in SLL

```
NodeT *deleteLast(ListT *listPtr)
{
    NodeT *q, *q1;
    q1 = NULL; /* initialize */
    q = listPtr->first;
    if ( q != NULL )
    {
        /* non-empty list */
        while ( q != listPtr->last )
        {
            /* advance towards end */
            q1 = q;
            q = q->next;
        }
        if ( q == listPtr->first )
        {
            /* only one node */
            listPtr->first = listPtr->last = NULL;
        }
        else
        {
            /* more than one node */
            q1->next = NULL;
            listPtr->last = q1;
        }
        // free( q );
    }
    return q;
}
```

3. Removing a node given by a *key* – shown in Listing 2.12.

Listing 2.12: Sample code for deleting a node with a given key from a SLL

```
int deleteByKey(ListT *listPtr, int givenKey)
{
        NodeT *q, *q1;
        q1 = NULL; /* initialize */
        q = first;
        /* search node */
        while ( q != NULL )
        {
          if ( q->key == givenKey ) break;
          q1 = q;
          q = q->next;
        }
        if ( q != NULL )
        {   /* found a node with supplied key */
          if ( q == first )
          { /* is the first node */
            first = first->next;
            free( q ); /* release memory */
            if ( first == NULL ) last = NULL;
          }
          else
          { /* other than first node */
            q1->next = q->next;
            if ( q == last ) last = q1;
            free( q ); /* release memory */
          }
          return 1; // success
        }
        return 0; // failure
}
```

Purging a Singly-linked List

For purging (removal of all elements) of a list, we have to remove each node of that list, as shown in Listing 2.13.

Listing 2.13: Sample code for purging a SLL

```
void purge(ListT *listPtr)
{
        NodeT *p;
        while ( listPtr->first != NULL )
        {
          p = listPtr->first;
          listPtr->first = listPtr->first->next;
          free( p );
        }
        listPtr->last = NULL;
        listPtr->count = 0;
}
```

Stacks

A *stack* is a special case of a singly-linked list which works according to LIFO[1] algorithm. Access is restricted to one end, called the *top* of the stack. A *stack* model is shown in Figure 2.3. Its operations are:

push – push an element onto the top of a stack;
pop – pop an element from the top of a stack;
top – retrieve the element located at the top of a stack;
purge – remove all elements from the stack. This can be done as explained in the previous paragraph.

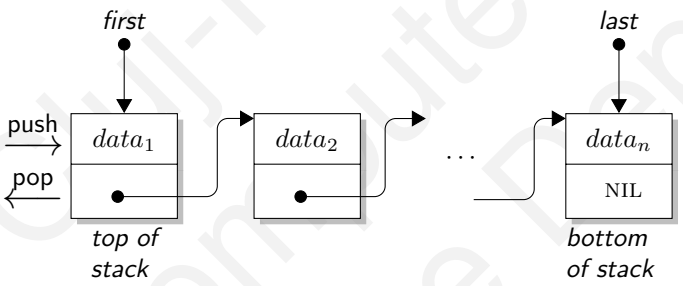

Figure 2.3: A stack model.

Queues

A *queue* is also a special case of a singly-linked list, which works according to FIFO[2] algorithm. Of the two ends of the queue, one is designated as the *front* – where elements are extracted (operation called *dequeue*), and another is the *rear*, where elements are inserted (operation called *enqueue*). A *queue* model is illustrated in Figure 2.4. The main operations are:

enqueue – place an element at the tail of a queue;
dequeue – take out an element form the front of a queue;
front – return, but not remove the element located at the front of a queue;
rear – return, but not remove the element located at the front of a queue;
purge – remove all elements from the queue. This can be done as explained at §2.3.

2.4 Laboratory Assignments

Study the provided code and use that code and code derived from it to solve the mandatory problems.

For all proposed problems, input and output data should reside in files whose names are given as command line arguments.

Here are some issues to think about:

[1] L[ast] I[n] F[irst] O[ut]
[2] F[irst] I[n] F[irst] O[ut]

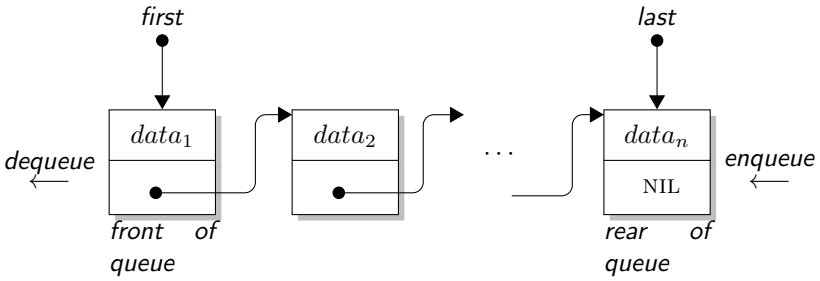

Figure 2.4: A queue model.

- What are the pros (arguments in favor) and cons (arguments against) for implementing a list using a header cell?
- What are the pros and cons for using static vs. dynamically allocated memory for storing a list?
- What are the limitations of singly-linked lists? What about the benefits?
- Why should we try to achieve a more general (abstract) implementation of a list instead of "solving" the problem at hand?
- What are the benefits of using a modular approach instead of a monolithic one in our implementation?
- How can we test our implementation?
- How can we generate test data to use in checking that our program works properly?

2.4.1 Mandatory Assignments

2.1. There is a garage where the access road can accommodate any number of trucks at one time. The garage is build such a way that only the last truck entered can be moved out (that is, a truck cannot overtake another while it is inside the garage). Each of the trucks is identified by a positive integer (a `truck_id`). Write a program to handle truck moves, allowing for the following commands:

a) `R`(truck_id); // place a truck on the road
b) `E`(truck_ id); // selected truck enters in the garage
c) `X`(truck_id); // selected truck exits the garage
d) `S`(`G` or `R`); // show the trucks in the garage (G) starting with the one closest to entry or on road(R)

If an attempt is made to get out a truck which is not the closest to the garage entry, an error message like the one shown after the next problem should be issued. .
I/O description. Input:

```
R(2)
R(5)
R(10)
R(9)
R(22)
S(R)
S(G)
E(2)
S(R)
S(G)
E(10)
E(25)
X(10)
X(2)
S(R)
S(G)
```

Output:

```
R:␣2␣5␣10␣9␣22
G:␣none
R:␣5␣10␣9␣22
G:␣2
error:␣25␣not␣on␣road!
error:␣10␣not␣at␣exit!
R:␣2␣5␣9␣22
```

```
G: 10
```

2.2. A similar problem, but now the road is circular and has two different entries/exits: one entry, and one exit. Trucks can get out only in the order they got in. Input data and output is presented in the same form as for the previous exercise.

2.4.2 Optional Assignments

2.3. Define and implement functions to operate on a singly linked list with a sentinel. The header cell is defined as follows:

```
typedef struct
{
    int count;
    NodeT *first; // always points to the sentinel
} ListT;
```

using the model illustrated in Figure 2.5. .
I/O description. Operations should be coded as: `del<number>`=delete `<number>` from list, `dcr`=delete the node at current position, `icr<number>`=insert after current position, `ist<number>`=insert `<number>` as first, `ila<number>`=insert `<number>` as last, `dst`=delete first cell, `dla`=delete last cell, `prt`=print list.

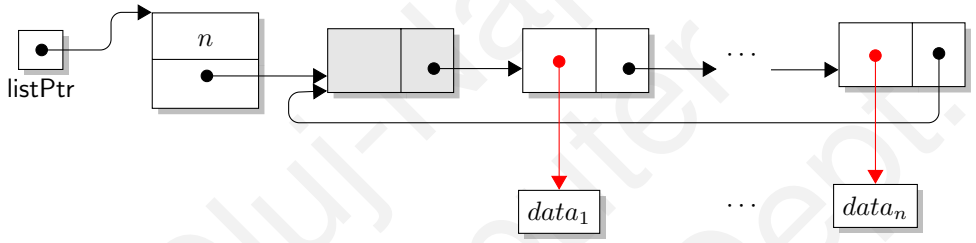

Figure 2.5: A model of a list for problem 2.3.

2.4. Implement a list as a static array containing pointers to information nodes allocated on the heap, using the model of Figure 2.6. .
I/O description. Operations should be coded as: `del<number>`=delete `<number>` from list, `ist<number>`=insert `<number>` as first, `ila<number>`=insert `<number>` as last, `dst`=delete first, `dla`=delete last, `prt`=print list.

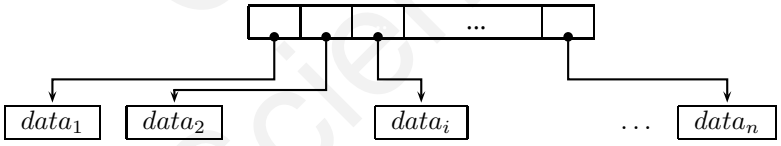

Figure 2.6: A model of a list for problem 2.4.

2.5. Create a singly linked list, ordered alphabetically, containing words and their occurrence frequency read from the input data. Print the list alphabetically. .
I/O description. Examples of input and output are given in Figure 2.7 Each item in the output is of the form $word : number\_of\_occurences$.

2.6. Topological sort. The elements of a set M are denoted by lowercase letters. Read pairs of elements $(x, y),\ x, y \in M$ meaning "element x precedes element y. List the elements of the set such that if an element precedes another, then it should be listed *before* its successors. .
I/O description. Input:

```
a,d
a,e
e,d
```

2.7. Write a program which creates two ordered lists, sorted ascending using the value of a numeric key, and then merges them together. .
I/O description. Input:

Input:

```
The␣quick␣brown␣fox␣jumps␣over␣the␣lazy␣dog.
```

Output (case sensitive):

```
The:1␣brown:1␣dog:1␣fox:1␣jumps:1␣lazy:1␣over:1␣quick:1␣the:1
```

Output (case insensitive):

```
brown:1␣dog:1␣fox:1␣jumps:1␣lazy:1␣over:1␣quick:1␣the:2
```

Figure 2.7: Example input and output for problem 2.5.

```
i1␣23␣47␣52␣30␣2␣5␣-2
i2␣-5␣-11␣33␣7␣90
p1
p2
m
p1
p2
```

Output:

```
1:␣-2␣2␣5␣23␣30␣47␣52
2:␣-11␣-5␣7␣33␣90
1:␣-11␣5␣2␣2␣5␣7␣23␣...
2:␣empty
```

Thus, "commands" accepted are: `i`n=insert onto list $n \in \{1, 2\}$, `p`n=print contents of list n, `m`=merge lists.

2.8. Imagine an effective dynamic structure for storing sparse matrices. Write operations for addition, subtraction, division, and multiplication of such matrices. .
I/O description. Input:

```
m1␣40␣40
(3,␣3,␣30)
(25,␣15,␣2)
m2␣40␣20
(5,␣12␣1)
(7␣14␣22)
m1+m2
m1*m2
```

, where `m1`=read elements for matrix `m1`, and the following triples are $(row, col, value)$ for that matrix. Reading ends when either another command or the end of file marker is encountered. **E.g.** `m1+m2`=add matrix 1 to matrix 2, and `m1*m2`=multiply matrix m1 with matrix m2. Output should be given in similar format, i.e triplets.

2.9. Imagine an effective dynamic structure for storing polynomials. Write operations for addition, subtraction, division, and multiplication of polynomials. .
I/O description. Input:

```
p1=3x^7+5x^6+22.5x^5+0.35x-2
p2=0.25x^3+.33x^2-.01
p1+p2
p1-p2
p1/p2
p1*p2
```

, Output:

```
<list␣of␣sum␣polynomial>
<list␣of␣difference␣polynomial>
q=<list␣of␣quotient,␣or␣0>
r=<list␣of␣remainder,␣or␣0>
<list␣of␣product␣polynomial>
```