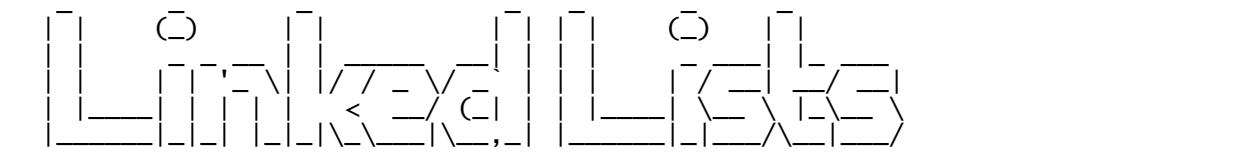

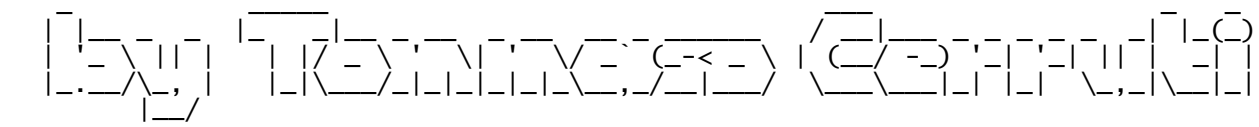

Topics:
- 1. Atomic operations
- 2. Common lists
- 3. Exercises
- 4. Exam-like exercises

1_ATOMIC_OPERATIONS

Definition:

```
typedef struct list_s list_t;
struct list_s{
        int key;
        ...
        list_t *next;
}
```

Allocation of a new node:

```
list_t *new_element(int key){
        list_t *e_ptr;
        e_ptr=malloc(sizeof(list_t));
        if (e_ptr==NULL){
                fprintf(stderr,"Memory allocation error.\n");
                exit(1);
        }
        return e_ptr;
}
```

Visiting the nodes:

```
list_t *p;
p=head;
while(p!=NULL){
        ...
        p=p->next;
}
```

Search 1:

```
list_t *p;
p=head;
while(p!=NULL){
        if (value==p->key){
                ...
        }
```

```
        else
                p=p->next;
}

Search 2:
list_t *p;
p=head;
while((p!=NULL)&&(p->key!=value))
        p=p->next;
if(p!=NULL){
        ...
}



Head extraction:
if (head!=NULL){
        p=head;
        head=head->next;
        ...
        free(p);
}

In-order extraction 1:
p=q->next;
q->next=p->next;
...
free(p);

In-order extraction 2:
p=q->next;
q->next=q->next->next;

Tail extraction:
p=q->next;
q->next=p->next;
//which is =NULL



Head insertion:
new=new_element(key);
new->next=head;
head=new;

In-order insertion:
new=new_element(key);
new->next=p->next;
p->next=new;

Tail insertion:
new=new_element(key);
new->next=p->next;
p->next=new;
//which is =NULL



Dispose list:
while(head!=NULL){
        p=head;
        head=head->next;
        ...
        free(p);
}
```

_____2_COMMON_LISTS_____

LIFO: Last-In First-Out (insert and extract elements from the same extreme --> head)

```
list_t *top;
int val, status;
top=NULL;
do{
        ...
        top=push(top,val);
        ...
        top=pop(top,&val,&status);
        ...
} while (...);

list_t *push(list_t *top, int val){
        list_t *new;
        new=new_element();
        new->key
        new->next=top;
        return new;
}

list_t *pop(list_t *top, int *val, int *status){
        list_t *old;
        if (top!=NULL){
                *status=SUCCESS;
                *val=top->key;
                old=top;
                top=top->next;
                free(old);
        }
        else
                *status=FAILURE;
        return top;
}
```

FIFO: First-In First-Out (insert and extract elements from different extreme --> tail insertion and head extraction)
Note:
Inserting on the tail requires to visit the entire list first(O(n)) which is not admissible
Thus two solutions: two pointers (head/tail) or circuital list (head=tail->next)

```
list_t *tail;
int val, status;
tail=NULL;
do{
        ...
        top=enqueue(tail,val);
        ...
        tail=dequeue(tail,&val,&status);
        ...
} while (...)

list_t *enqueue (list_t *tail, int val){
        list_t *new;
        new=new_element();
        new->key=val;
        if (tail==NULL){
```

```
                tail=new;
                tail->next=tail;
        }
        else{
                new->next=tail->next;
                tail->next=new;
                tail=new;
        }
        return tail;
}

list_t *dequeue (list_t *tail, int *val, int *status){
        list_t *old;
        if (tail!=NULL){
                *status=SUCCESS;
                if (tail==tail->next){
                        *val=tail->key;
                        free(tail);
                        tail=NULL;
                }
                else{
                        old=tail->next;
                        *val=old->key;
                        tail->next=old->next;
                        free(old);
                }
        }
        else
                *status=FAILURE;
        return tail;
}
```

ORDERED: Linked list maintained sorted (either by increasing or decreasing key values)
Because we need to maintain the list sorted, typically insertions are done in the middle

```
list_t *head;
int val, status;
head=NULL;
do{
        ...
        top=insert(top,val);
        ...
        search(head,val);
        ...
        top=extract(top,val,&status);
        ...
} while (...);

void search(list_t *head, int val){
        list_t *p;
        p=head;
        while(p!=NULL && p->val<val){
                p=p->next;
        }
        if (p!=NULL && p->val==val){
                //found
        }
        return
}

list_t *insert(list_t *head, int val){
        list_t *p, *q=head;
        p=new_element();
        p->val=val;
        p->next=NULL;
```

```
        if (head==NULL || val<head->val){
                //head insertion
                p->next=head;
                return p;
        }
        while (q->next!=NULL && q->next->val<val){
                q=q->next;
        }
        p->next=q->next;
        q->next=p;
        return head;
}

list_t *extract(list_t *head, int val, int *status){
        list_t *p, *q=head;
        if (head==NULL){
                *status=FAILURE;
                return head;
        }
        if (val==head->val){
                //head extraction
                p=head->next;
                free(head);
                *status=SUCCESS;
                return p;
        }
        while(q->next!=NULL && q->next->val<val)
                q=q->next;
        if (q->next!=NULL && q->next->val==val){
                p=q->next;
                q->next=p->next;
                free(p),
                *status=SUCCESS;
                return head;
        }
        *status=FAILURE;
        return head;
}
```

DOUBLE-LINKED LIST: Possibility to move in both direction

```
typedef struct list_s list_t;
struct list_s{
        int key;
        ...
        list_t *left, *right;
}
```

LIST OF LISTS: Each element of the list is a list itself

```
typedef struct list_s1 list_t1;
typedef struct list_s2 list_t2;
struct list_s1{
        int key;
        ...
        list_t1 *next;
        list_t2 *right;
}
struct list_s2{
        int key;
        ...
        list_t2 *next;
}
```

_____3_EXERCISES_____

A. List Inversion

Problem: Given a simple linked list, invert all elements of such a list such that
the first becomes the last one and the last one becomes the first one.

Example:
Input list: head1 --> 30 --> 17 --> 21 --> 9
Output list: head2 --> 9 --> 21 --> 17 --> 30

Strategy: while(head1!=NULL){pop(head1); push(head2)}

Implementation:

```
list_t *list_reverse(list_t *head1){
        list_t *tmp1, *head2;
        head2=NULL;
        while(head1!=NULL){
                tmp1=head1;
                head1=head1->next;
                tmp1->next=head2;
                head2=tmp1;
        }
        return head2;
}
```

B. List Sort

Problem: Given a simple linked list, sort all elements in ascending order

Example:
Input list: head1 --> 30 --> 17 --> 21 --> 9
Output list: head2 --> 9 --> 17 --> 21 --> 30

Strategy: extract all elements from the first list (head extraction), and insert elements
into the second list (in-order insertion) and finally substitute first list with the second one.

```
//using the functions previously defined with few modifications
list_t *list_sort(list_t *head1){
        list_t *tmp, *head2;
        head2=NULL;
        while(head1!=NULL){
                tmp=pop(&head1);
                head2=insert(head2,tmp);//insert includes in-order insertion
        }
        return head2;
}
```

_____4_EXAM-LIKE_EXERCISES_____

Exercises taken from Stefano Quer's book{}

1.2
Write a function able to inser some registry data i the correct position in an ordered list. Data include surname and name of a person, both strings of max length 20.
Use the surname as primary ordering key and the name as the second.
The prototype is int list_insert_in_order(list_t **headP, char *surname, char *name);
returning false when person is already present or true when it is inserted.

Solution:

```
typedef struct node_s node_t;

struct node_s{
        char *lastname;
        char *name;
        node_t *next;
}

list_t *create_node(void);

int list_insert_in_order(list_t **headP, char *surname, char *name){

        list_t *node, *p, *head;
        //first check if the person is present
        head=*headP;
        p=head;
        while(p!=NULL && (strcmp(surname,p->surname)==0) &&
(strcmp(name,p->name)==0))
                p=p->next;

        //the person is present
        if (p!=NULL)
                return 0;

        //the person is not present, create the node and insert it in order
        node=create_node;
        node->surname=strdup(surname);
        if (node->surname==NULL){
                fprintf(Stderr,"Memory allocation error");
                exit(1);
        }
        node->name=strdup(name);
        if (node->name==NULL){
                fprintf(Stderr,"Memory allocation error");
                exit(1);
        }
        p=head;

        if (p==NULL || strcmp(surname,p->surname)<0 ||
strcmp(surname,p->surname)==0 &&
strcmp(name,p->name)<0){
                node->next=head;
                (*headP)=node;
                return 1;
        }

        while(p->next!=NULL && (strcmp(surname,p->next->surname)>0) &&
((strcmp(surname,p->next->surname)==0) && (strcmp(name,p->next->name)>0)){
                p=p->next;
        }
        node->next=p->nextM
        p->next=node;
        return 1;
```

```
}

list_t *create_node(void){
        list_t *node;
        node=malloc(sizeof(list_t));
        if (node==NULL){
                fprintf(Stderr,"Memory allocation error");
                exit(1);
        }
        return node;
}
```

3.2
A bi-linked list of integers is given (node_t is the basic data type of each element).
Implement the following two functions:
- void list_insert(node_t **left, node_t **right, int key, int leftRight);
to insert the value key on the left/right of the extreme of the list if leftRight is 0/1
- void list_write(node_t *left, node_t *right, int leftRight);
to print-out the content of the entire list visiting it from left/right if left leftRight is 0/1

Solution:

```
typedef struct node_s node_t;

struct node_s{
        int key;
        node_t *left;
        node_t *right;
}

list_t *create_node(int key);

void list_insert(node_t **left, node_t **right, int key, int leftRight){
        node_t *node;
        node=create_node(key);
        if (!leftRight){
                //insertion on the left
                node->right=(*left);
                node->left=NULL;
                if ((*left)!=NULL)
                        (*left)->left=node;
                (*left)=node;
                if ((*right)==NULL)
                        (*right)=(*left);
        }
        else{
                //insertion on the right
                node->left=(*right);
                node->right=NULL;
                if ((*right)!=NULL)
                        (*right)->right=node;
                (*right)=node;
                if ((*left)==NULL)
                        (*left)=(*right);
        }
        return;
}

void list_write(node_t *left, node_t *right, int leftRight){
        node_t *start;
        if (!leftRight){
                //write from the left
                start=left;
                while(start!=NULL){
                        printf("%d ",start->key);
                        start=start->right;
```

```
                }
        }
        else{
                //write from the right
                start=right;
                while(start!=NULL){
                        printf("%d ",start->key);
                        start=start->left;
                }
        }
        return;
}

list_t *create_node(int key){
        list_t *node;
        node=malloc(sizeof(list_t));
        if (node==NULL){
                fprintf(Stderr,"Memory allocation error");
                exit(1);
        }
        node->key=key;
        return node;
}
```
