UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

S07: Data Structure

A. Schaller

16-896-375

Prof.: Dr. Luggen Michael

April 18, 2021

1 Some Linked List Data Structures

Question 1. *Using struct, declare the data structure of the following structures:*

1. *a linked list*

```
struct node {
    int data;
    struct node * next;
};
```

Or replace the `int data` with the type expected to be in the structure, or replace it with a pointer, which will be using the same amount of memory to point to any data.

2. *a doubly linked list*

```
struct node {
    int data;
    struct node * previous;
    struct node * next;
};
```

Same comment as the previous linked list.

3. *a linked binary tree*

```
struct node {
    int data;
    struct node * left;
    struct node * right;
};
```

Same comment.

2 Pointer Manipulation

Question 1. *Define a data structure that corresponds to the sketch in Fig.1., and implement the function void swap_ptr() allowing to swap the two top elements as showed in Fig.1..*

Remark – To simplify the implementation of swap_ptr(), we assume that the data structure contains at least 2 nodes, i.e. no error checking has to be performed. Beware that your solution should also work when the structure contains only 2 nodes.

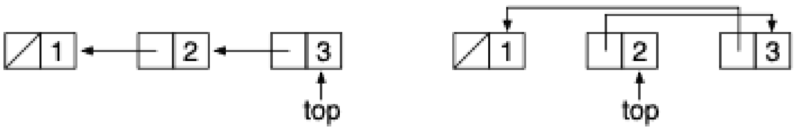


Figure 1: A data structure before and after 'swap_ptr();'

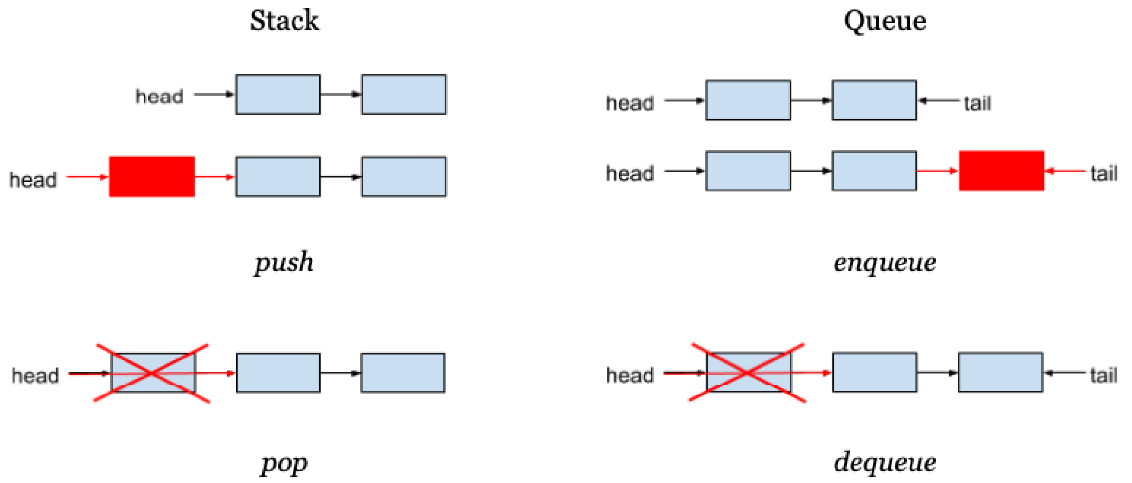
The *Fig.1.* represent a *simple linked list* and the swap operation is equivalent to swapping the payload of the head with its previous node's payload (even though the name isn't really representative of the behavior).

```
struct node {
    int data;
    struct node * next;
};

void swap_ptr(struct node * head) {
    int headData = head->data;
    int nextData = head->next->data;

    head->data = nextData;
    head->next->data = headData;
}
```

3 Queue

Question 1. *Study the linked_list_stack.c[1] source code and transform it to implement a queue. A queue is a data structure in which objects are accessed in FIFO (First In First Out) order. New objects are inserted at the end of the queue (enqueue). Object is removed from the beginning (dequeue).*

Hint – to easily access the end of the queue, you need to store an additional pointer to the last element of your linked list. Create a structure struct Queue which will store the head node and tail node of the list.




```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node * next;
};

struct Queue {
    struct Node * head;
    struct Node * tail;
};

void printLinkedlist(struct Node * p);

void enqueue (struct Queue * q, struct Node * newTail) {
    if (q->tail) {
        q->tail->next = newTail;
    }
    // if we don't have a tail, we neither have a head, unless external
    // change
    else {
        q->head = newTail;
    }
```

[1] https://unifr.coursc.ch/7/linked_list_stack.c (visited on April 2021)

```
    // If we allow enqueuing a tail with multiple nodes, then we have
    // to look for the deepst tail
    // -- Possible improvement? --
    q->tail = newTail;
}

void enqueue_value (struct Queue * q, int value) {
    struct Node * newNode = malloc(sizeof(newNode));
    newNode->data = value;
    enqueue(q, newNode);
}

struct Node * dequeue (struct Queue * q) {
    struct Node * head = q->head;

    if (head) {
        q->head = head->next;
        head->next = 0;
    }

    return head;
}

int main () {
    struct Queue * q = malloc(sizeof(q));
    enqueue_value(q, 2);
    enqueue_value(q, 5);
    enqueue_value(q, 10);

    printLinkedlist(q->head);

    struct Node * node = dequeue(q);
    printf("Dequeued element: %d (next: %p)\n", node->data, node->next);

    struct Node last = { 15, 0 };
    enqueue(q, &last);

    printLinkedlist(q->head);
}
```

4 Graph

Question 1. *Study the minimal_tree.c[2] source code. Suggest a new data structure to be able to express directed graphs instead of trees:*

- *Modify the struct node structure and the newNode function.*

```
// 1. modifiy the struct
struct LinkedNode {
    struct GraphNode * target;
```

[2] https://unifr.coursc.ch/7/minimal_tree.c (visited on April 2021)

```c
    struct LinkedNode * next;
};

struct GraphNode {
    int data;
    struct LinkedNode * directed_relations;
};

// 2. modifiy the newNode function
struct GraphNode * new_graph_node (int data) {
    struct GraphNode * newNode = malloc(sizeof(newNode));
    newNode->data = data;
    newNode->directed_relations = 0;
    return newNode;
}
```

- *Write an additional function connect which allows to connect two arbitrary nodes in your graph.*

```c
struct LinkedNode * new_linked_node (struct GraphNode * g) {
    struct LinkedNode * newNode = malloc(sizeof(newNode));
    newNode->target = g;
    newNode->next = 0;
    return newNode;
}

void connect (struct GraphNode * source, struct GraphNode * target) {
    // should maybe also check if the target is not in the
       directed_relations list
    struct LinkedNode * last_relation = source->directed_relations;
    struct LinkedNode * new_relation_target = new_linked_node(target);
    if (last_relation) {
        while (last_relation->next) {
            last_relation = last_relation->next;
        }

        last_relation->next = new_relation_target;
    }
    // no directed_relations yet
    else {
        source->directed_relations = new_relation_target;
    }
}
```

A running example using those different structures and functions:

```c
void print_relations (struct GraphNode * node) {
    int data = node->data;
    printf("%d", data);
    if ( ! node->directed_relations) {
        // has no directed relation
        printf(" has no directed relations\n");
    }
```

```c
    struct LinkedNode * r = node->directed_relations;
    printf(" relations:");
    while (r) {
        printf(" %d", r->target->data);
        r = r->next;
    }
    printf("\n");
}

/*
* Example graph to build:
*
*               5
*             / |
*            /  |
*           /   |
*          v    v
*          2-+  1  +-->4
*          \ +--+--+ / ^
*           \   |   / /
*            \  |  / /
*             \ | / /
*              vvv /
*               3
*/

int main () {
    struct GraphNode * top = new_graph_node(5);
    struct GraphNode * left = new_graph_node(2);
    struct GraphNode * middle = new_graph_node(1);
    struct GraphNode * right = new_graph_node(4);
    struct GraphNode * bottom = new_graph_node(3);

    connect(top, left);
    connect(top, middle);

    connect(left, right);
    connect(left, bottom);

    connect(middle, bottom);

    connect(right, bottom);

    connect(bottom, right);

    print_relations(top);
    print_relations(left);
    print_relations(middle);
    print_relations(right);
    print_relations(bottom);
}
```

And its output:

```
5 relations: 2 1
2 relations: 4 3
1 relations: 3
4 relations: 3
3 relations: 4
```

5 Project P01: Linked Data In-Memory Store

Question 1. *Think about the data structure you want to use for your storage. Describe it, its advantages and potential pitfalls.*

this chapter is in common for the whole group

5.1 Definition of basic data structures used for our store

```
vector
```

List of variable size. Operations:

- insertion: amortized $O(1)$
- access: $O(1)$

```
hashmap
```

Map from keys to values. Operations:

- insertion: $O(1)$ key hashes
- access: $O(1)$ key hashes + key comparisons (assuming good hash function)

5.2 Structure of the store

We want to answer 8 different kind of queries: `{SPO}`, `{SP}`, `{SO}`, `{PO}`, `{S}`, `{P}`, `{O}`, `{}`

We use a separate hashmap for each of those type of queries. The key is the known values (for example, `S` and `P` for the type `{SP}`). The value is a vector of pointers to the matching triples.

The triples themselves are stored in the heap.

```
access (match(s, p, o, result))
```

1. Determine the type of the query by looking at the parameters `"s"`, `"p"` and `"o"` (for example `{SP}`) $\rightarrow O(1)$

2. Access the corresponding hashmap using the known values (for example `"s"` and `"p"`) to retrieve the list (vector) of all matching triples $\rightarrow O(1)$ *key hashes + key comparisons (assuming good hash function)*
3. Access the retrieved vector to get the desired triple (at the index given by the parameter "result") $\rightarrow O(1)$

The "key comparison" and "key hash" operations time complexity grows linearly with the size of the known data (strings) (for example `"s"` and `"p"`).

$\Rightarrow$ Overall time complexity: $O(n)$ where n is the size of the known data (**does not grow with the number of triples in the store**).

```
insertion (insert(s, p, o))
```

1. Allocate memory to store the new triple in the heap
2. For all 8 type of queries $\rightarrow O(1)$
   (a) Access the corresponding hashmap using the known values (for example `"s"` and `"p"`) to retrieve the list (vector) of all matching triples $\rightarrow O(1)$ key hashes + key comparisons (assuming good hash function)
   (b) Insert the pointer to the new triple in the retrieved vector $\rightarrow$ amortized $O(1)$

The "key comparison" and "key hash" operations time complexity grows linearly with the size of the known data (for example `"s"` and `"p"`).

$\Rightarrow$ Overall time complexity: $O(n)$ where n is the size of the given data (strings) (**does not grow with the number of triples in the store**).

5.3 Advantages and pitfalls

+ very fast insertion and access (assuming good hash function)

– requires the implementation of the basic data structures (vector and hashmap)
– if implemented, deletion of arbitrary triples will have worst-case $O(n)$ time complexity where n is the number of triples in the store (potential solution: do not really delete the triple, mark it with a "deleted" flag, and the caller of "match" is responsible to check if the triple is deleted and does not use it in that case)