Group 14

Assignment 2 - ALDA

(1) (Week 3) List Abstract Data Type (ADT) and Linked List. Note that your main code must test all implemented functions.

(a) Implement the List ADT using a linked list that you implement from scratch. Use the code provided in Week 2 as a starting point, and write the main function that tests your implementation. Your linked list should support the following operations and should store a generic element: insertion at the beginning, middle (given a position), and end of the list; deletion of an element from the beginning, middle (given a position), and end of the list; check if a given element exists in the list; and traversal of the list to print all elements.

(b) Add, implement, and test, a function to the List ADT to reverse the list.

a) *The code for this exercise can be found in the folder Exercise 1.*

Overview

We implemented a generic List (ADT) using a doubly linked list with sentinel head and tail nodes. The list stores elements of any type T and supports all operations requested in the assignment. The implementation is split as a header (list.h) with template declarations and a .tpp file (list.tpp) with the template definitions so the compiler can see them.

Double vs single linked list

We use a doubly linked list when we need traversal in both directions.
It makes insertion and deletion faster since we don't need the previous node.
The trade-off is extra memory for the extra pointer.

Data structure

Nodes hold data, next and prev and the sentinels: head and tail are dummy nodes.
The start state involves head->next = tail, tail->prev = head, size theSize = 0.

b) We added the function void List<T>::reverse() in list.tpp and declared in list.h. It reverses the entire doubly linked list with sentinels (head, tail) in-place. *This function can be found in line 209.*

Algorithm

1. Start at head and iterate through every node, including our two sentinels
2. For each node, we swap its next and prev pointers
3. When the sweep is done, the direction of all links are flipped. (reversed)

(2) (Week 3) Implement a Stack using an ordinary array. The stack should be initialized with a size parameter that defines the initial size of the stack. The default constructor should create a stack with 100 elements as the initial size. When `push(...)` is called and the stack is full, a new array should be allocated that have the double size, the content of the old array copied to the new one and the old array deleted. Analyze the running time of the stack operations `push` and `pop` in terms of the N elements it stores.

We interpreted this exercise as a challenge to take an ordinary array and make it dynamic instead of using other types of dynamic containers such as vectors or arraylists. In the file stack.hpp we've implemented the basic stack functions such as push, pop, isEmpty, size, capacity and top and then in main.cpp (for exercise 2 & 3 we wrote some test cases to test if our program worked as intended. Additional and more code descriptive comments can be found as comments in stack.hpp.

The Big O running time complexity analysis can be seen in the code for the concrete functions.

(3) (Week 3) Write an implementation of `Queue` using `Stack`(s). The implementation must be a template and use the Stack ADT presented in the course. There is no requirement that the implementation is efficient, but you cannot use the internal structure of the stack, nor an iterator for the stack.

Our implementation of the queue is in our repository under exercise 2 and 3 as "queue.hpp". We interpret this task as we have to use the implementation of stack from exercise 2 to implement the queue functions queue, ~~queue, enqueue, dequeue, front and empty. We are using a template, making it possible to use multiple data types, when initializing a queue. Afterwards making a main.cpp to test all the functions operate properly.

(4) (Week 4) Let T by a hash-table of size 7 with the hash function $h(x) = x \, mod \, 7$. Write down the entries of T after the keys 5, 28, 19, 15, 20, 33, 12, 17, 33 and 10 have been inserted using

(a) chaining

(b) linear probing

(c) quadratic probing

What is the load-factor(λ) in the three cases

a) Chaining

Collisions go into a linked list (or chain) at the same index.

Index	Chain(list of keys)
0	28
1	15

2	
3	17, 10
4	
5	5, 19, 33, 12
6	20

n = number of stored elements (all of them, even if many in one chain).
m = number of slots (buckets).

Load factor = n / m = 9 / 7

Explanation:
5 % 7 = 5 -> goes to index 5.
28 % 7 = 0 -> goes to index 0.
19 % 7 = 5 ->goes to index 5.
15 % 7 = 1 -> goes to index 1.
20 % 7 = 6 -> goes to index 6.
33 % 7 = 5 -> goes to index 5.
12 % 7 = 5 -> goes to index 5.
17 % 7 = 3 -> goes to index 3
33 % 7 = 5 ->IGNORED Because we assume in the exercise that we have a set hash like table, so duplicates are ignored (Is not specified in exercise).
10 % 7 = 3 -> goes to index 3.

b) Linear probing

Collisions are resolved by moving forward one step at a time until an empty slot is found.

Index	Key
0	28
1	15
2	20
3	33
4	12
5	5
6	19

n = number of stored elements (all of them, even if many in one chain).
m = number of slots (buckets).

Load factor = n / m = 7 / 7 = 100%

Explanation:
5 % 7 = 5 -> goes to index 5.
28 % 7 = 0 -> goes to index 0.
19 % 7 = 5 -> idx 5 (taken) -> goes to index 6.
15 % 7 = 1 -> goes to index 1.
20 % 7 = 6 -> idx 6 (taken) -> 0 (taken) -> 1 (taken) -> goes to index 2.
33 % 7 = 5 -> idx 5 -> 6 -> 0 -> 1 -> 2 -> goes to index 3.
12 % 7 = 5 -> … -> goes to index 4
17 % 7 = 3 -> fails
33 % 7 = 5 -> fails
10 % 7 = 3 -> fails

We assume the hash table has a fixed size of 7, this means 17, 33 and 10 are not inserted as there are no empty slots(or triggers a rehash, but here we don't do that).

c) Quadratic probing

Collisions resolved by checking slots with offset **i^2**:

Index	Key
0	0
1	15
2	33
3	20
4	17
5	5
6	19

Explanation:
5 % 7 = 5 -> goes to index 5.
28 % 7 = 0 -> goes to index 0.
19 % 7 = 5 -> 5 (taken) -> goes to index 6.
15 % 7 = 1 -> goes to index 1.

20 % 7 = 6 -> 6 (taken) -> 0 (i = 1, taken) -> goes to index 3 (i = 2)
33 % 7 = 5 -> 5 (taken) -> 6 (i = 1, taken) -> goes to index 2 (i = 2)
12 % 7 = 5:
cycles through {5, 6, 2, 0} -> fails as it never goes to index 4
17 % 7 = 3 -> 3 (taken) -> goes to index 4
33 % 7 = 5 -> fails
10 % 7 = 3 -> fails

Load factor = n / m = 7 / 7 = 100 %

(5) (Week 4) Implement a `Set` using either a `Queue`, `List` or `Stack` ADT as presented in the course. There is no requirement that the implementation is efficient, but you cannot use the internal structure of the ADT, nor an iterator for the ADT

The code for this exercise can be found in the Exercise 5 folder.

Implementation
We implemented generic Set<T> using the list ADT only. The set enforces uniqueness, which means no duplicates and exposes the standard operations:

- bool add(const T& x) - insert x only if it's not already present
  uses list.contains(x) -> if false, list.push_front(x)

- bool remove(const T& x) - deletes x if it exists
  uses list.contains(x) + list.delete_element(x)

- bool contains(const T& x) const
  uses list.contains(x)

- int size() const - number of elements
  uses list.size()

- bool isEmpty() const - emptiness check
  uses list.empty()

- void clear() - remove all elements
  uses list.clear()

- void print() const - print all elements
  uses list.print_all()

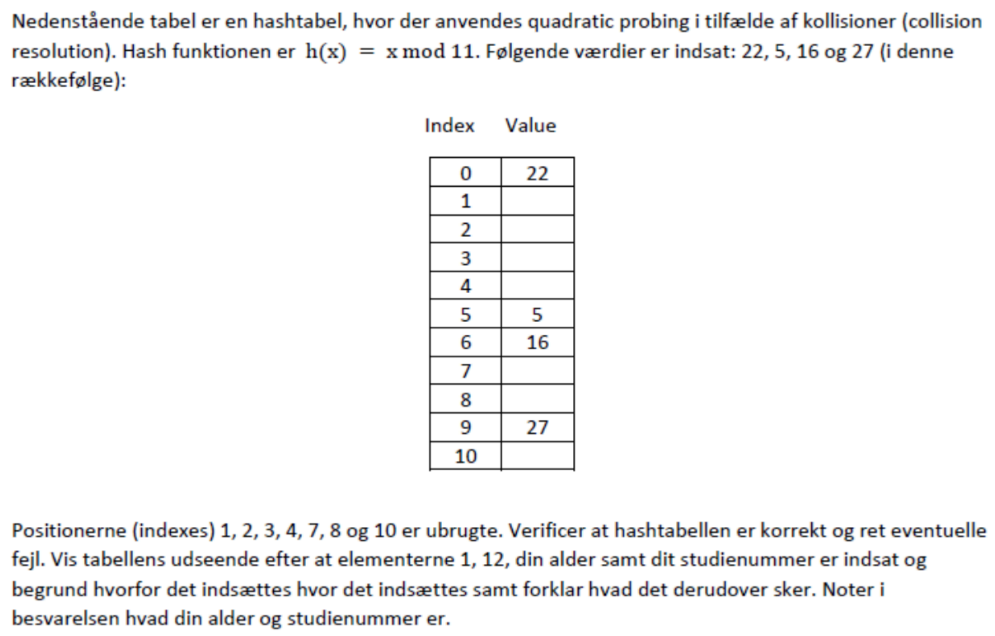
Happy testing!

(6) (Week 4) Implement a `Dictionary` or `Map` using an STL vector. The element of the vector (for the implementation) must be an STL pair representing the (key, value). You can find an example of how to use pair here: https://www.geeksforgeeks.org/pair-in-cpp-stl/

The code for this exercise can be found in the Exercise 6 folder.

We implemented the dictionary using the provided vector methods and constructor/destructor and pairs with keys and values. The explanation of the functions can be read in the commentary of the code in Dictionary.h.

(7) (Week 4) An old exam question



Nedenstående tabel er en hashtabel, hvor der anvendes quadratic probing i tilfælde af kollisioner (collision resolution). Hash funktionen er $h(x) = x \bmod 11$. Følgende værdier er indsat: 22, 5, 16 og 27 (i denne rækkefølge):

Index	Value
0	22
1	
2	
3	
4	
5	5
6	16
7	
8	
9	27
10	

Positionerne (indexes) 1, 2, 3, 4, 7, 8 og 10 er ubrugte. Verificer at hashtabellen er korrekt og ret eventuelle fejl. Vis tabellens udseende efter at elementerne 1, 12, din alder samt dit studienummer er indsat og begrund hvorfor det indsættes hvor det indsættes samt forklar hvad det derudover sker. Noter i besvarelsen hvad din alder og studienummer er.

Quadratic probing:

Collisions resolved by checking slots with offset **i²**:
22, 5, 16, 27

Index	Value
0	22
1	
2	
3	

4	
5	5
6	16
7	
8	
9	27
10	

Verification of keys:

22 % 11 = 0 -> goes to index 0.
5 % 11 = 5 -> goes to index 5.
16 % 11 = 5 -> goes to index 5 -> goes to index 6.
27 % 11 = 5 -> idx 5 (taken) -> idx 6 (taken) -> goes to index 9.

We can then conclude that the hash table is correct.

We assume that the previous values still exist in the hash table.

Numbers chosen (in order):
1, 12
Age: 24
Study-ID: 202406052

Updated hash table:

Index	Value
0	22
1	1
2	12
3	
4	
5	5
6	16

7	202406052
8	
9	27
10	

22 % 11 = 0 -> goes to index 0.
5 % 11 = 5 -> goes to index 5.
16 % 11 = 5 -> goes to index 5 -> goes to index 6.
27 % 11 = 5 -> idx 5 (taken) -> idx 6 (taken) -> goes to index 9.
1 % 11 = 1 -> goes to index 1
12 % 11 = 1 -> idx 1 (taken) -> goes to index 2 (i = 1)
24 % 11 = 2 -> idx 2 (taken) -> goes to index 3 (i = 1)
202406052 % 11 = 2 -> idx 2 (taken) -> idx 3 (i = 1, taken) -> idx 6 (i = 2, taken)
-> idx 0 (i = 3, taken) -> goes to index 7
Load factor = n / m = 7 / 11 = 0.63 = 63%