Lecture review

Resizable Arrays and Linked Lists

In data structures, resizable arrays and linked lists are two common ways to store collections of elements dynamically.

- A resizable array (also known as a dynamic array) starts with a fixed size but can automatically grow when it becomes full. When resizing occurs, a new, larger array is allocated, and all elements are copied to it. This allows efficient storage while maintaining sequential memory access.
  - The provided `resizable_array.cpp` implements an array using dynamic memory.
- A linked list consists of nodes, where each node stores a data value and a pointer to the next node in the sequence. Unlike arrays, linked lists do not require continuous memory allocation and allow elements to be inserted or removed without shifting other elements. However, accessing elements requires traversal from the beginning of the list.
  - The provided `linked_list.cpp` implements a linked list.

These two structures are commonly used to implement stacks and queues.

Stack and Queue

Stack and *queue* are two important ADTs. A stack stores items in a last-in, first-out (LIFO) manner, whereas a queue stores items in a first-in, first-out (FIFO) manner. Following table summarizes the operations of a stack and a queue.

Stack	Queue	Description
`push(x)`	`enqueue(x)`	Insert an item x
`pop()`	`dequeue()`	Remove and return an item (most recently inserted item for stack and least recently inserted one for queue)
`empty()`	`empty()`	Return true if the stack/queue is empty
`size()`	`size()`	Return the number of items in the stack/queue

We have seen two implementations of stacks and queues in the lecture: `RAStack` and `RAQueue` using *resizable arrays*, and similarly `LLStack` and `LLQueue` using *linked lists*. Note that from the perspective of the user, the two implementations of stack (respectively queue) are indistinguishable as both provide the identical public interface. The following files are provided for your reference:

- `ra_stack.cpp` and `ll_stack.cpp` implement a stack using resizable array and linked list respectively.
- `ra_queue.cpp` and `ll_queue.cpp` implement a queue using resizable array and linked list respectively.

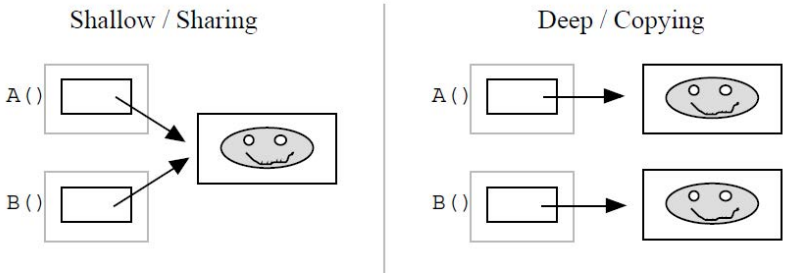
Copy constructor: Shallow vs Deep copy

A *copy constructor* is a special constructor that initializes a new object from an existing object. It is used to create a new object as a copy of an existing object. When copying objects that contain pointers, (as the above implementations of stack and queue do), we need to be careful about how we want to make a copy. For instance, consider the class `RAStack` that contains a pointer `data` to an array of integers. The copy constructor of `RAStack` can be implemented in following two ways:

```
RAStack(const RAStack& rhs)
  : arr(rhs.arr),
    sz(rhs.sz),
    cap(rhs.cap) {
}
```

```
RAStack(const RAStack& rhs)
  : arr(new string[rhs.cap]),
    sz(rhs.sz),
    cap(rhs.cap) {
  for (int i = 0; i < sz; ++i)
    arr[i] = rhs.arr[i];
}
```

On the left is the *shallow copy* implementation of the copy constructor, and on the right is the *deep copy* implementation. By shallow copy, we mean that the copy constructor copies the pointer but not the object that the pointer points to. Whereas deep copy means that the copy constructor allocates a new array and copies the elements of the array pointed to by `rhs.arr` to the new array.
Note that with shallow copy, the two objects `*this` and `rhs` will share the same array. So if we modify the array of one object, the other object will also be affected. This is not the case with deep copy as both stack objects will have their own copy of the array. So naturally, deep copy is more expensive than shallow copy as it involves allocating a new array and copying the elements of the array. But it is also safer as the two objects will not share the same array.
Following figure illustrates the difference between shallow and deep copy:



Copy Assignment Operator

The assignment operator is used to assign a new value to an existing variable. This operator can be overloaded to provide a specific implementation of the assignment for objects of a class. Such an implementation is called a copy assignment operator. For example, consider the following implementation of the copy assignment operator for `RAStack`:

```
RAStack& operator=(const RAStack& rhs) {
  if (this != &rhs) {
    delete[] arr;
    arr = new string[rhs.cap];
    sz = rhs.sz;
    cap = rhs.cap;
    for (int i = 0; i < sz; ++i)
      arr[i] = rhs.arr[i];
  }
  return *this;
}
```

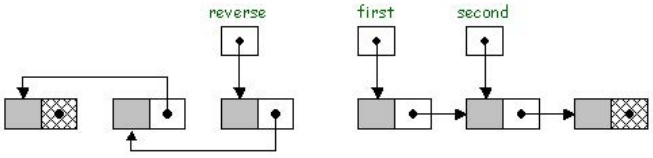
The concept of shallow and deep copy also applies to the copy assignment operator. The above implementation of the copy assignment operator is a deep copy implementation.

Lab exercises

Exercise 1

1. *Reverse a linked list.* Write a function for the provided linked list class that takes the first `Node` in a linked list as an argument, and reverses the list, returning the first `Node` in the result.

   Solution: To accomplish this, we maintain references to three consecutive nodes in the linked list, `reverse`, `first`, and `second`. At each iteration we extract the node first from the original linked list and insert it at the beginning of the reversed list. We maintain the invariant that `first` is the first node of what's left of the original list, `second` is the second node of what's left of the original list, and `reverse` is the first node of the resulting reversed list.

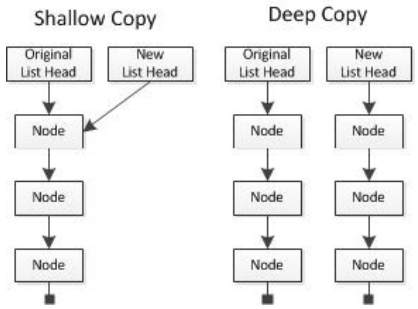


2. Write a recursive function to print a linked list in reverse order. You may not use any loops or iteration.

Exercise 2

Copy constructor for a stack. Create a new constructor for class `LLStack` (linked-list implementation of stack) so that `LLStack t {s};` makes `t` reference a new and independent copy of the stack `s`. That is, a deep copy of linked list pointed by `s.first` and make `t.first` point to the new list.

With deep copy, you should be able to push and pop from either `s` or `t` without influencing the other, demonstrate this by writing client code in the main function.



Exercise 3

Write a Stack client that reads a string of parentheses, square brackets, and curly braces from standard input and uses a stack to determine whether they are properly balanced. For example, your program should print true for `[()]{}{[()()]()}` and false for `[(])`.

Exercise 4

Josephus problem. In the *Josephus problem* from antiquity, n people are in dire straits and agree to the following strategy to reduce the population. They arrange themselves in a circle (at positions numbered from 0 to $n-1$) and proceed around the circle, eliminating every m-th person until only one person is left. Legend has it that Josephus figured out where to sit to avoid being eliminated.

Write a Queue client that takes two integer inputs `m` and `n` and prints the order in which people are eliminated (and thus would show Josephus where to sit in the circle).

- *Example 1:*
  Input: $n = 5$ and $m = 2$
  Output: The safe position is 2
  Explanation: Firstly, the person at position 1 is eliminated, then person at position 3 is eliminated, then person at position 0 is eliminated. Finally, the person at position 4 is eliminated. So the person at position 2 survives.
- *Example 2:*
  Input: $n = 7$ and $m = 3$
  Output: The safe position is 3
  Explanation: The persons at positions $2, 5, 1, 6, 4, 0$ are eliminated in order, and the person at position 3 survives.

Solution: We use a queue to store the positions of the people. Do this m-1 times: dequeue a person from the queue and enqueue them back at the end of the queue. Then the next person at front of queue, to be dequeued is eliminated. We repeat this process until only one person is left in the queue.