Linked Memory

head_ → C → S → 2 → 2 → 5 → Ø
template <typename T>
T List<T>::remove(unsigned index) {
}

#pragma once

template <typename T>
class List {
    public:
        /* ... */
    private:

};
Array Implementation

Array:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>S</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>
Array Implementation

insertAtFront:

<table>
<thead>
<tr>
<th>C</th>
<th>S</th>
<th>2</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
</tr>
</tbody>
</table>
Resize Strategy: +2 elements every time
Resize Strategy: x2 elements every time
Resize Strategy: x2 elements every time
## Array Implementation

<table>
<thead>
<tr>
<th></th>
<th>Singly Linked List</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert/Remove at front</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert at <em>given</em> element</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove at <em>given</em> element</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert at <em>arbitrary</em> location</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove at <em>arbitrary</em> location</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
std::vector
std::vector

Defined in header `<vector>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```

namespace pmr {
    template <class T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}

1) std::vector is a sequence container that encapsulates dynamic size arrays.
2) std::pmr::vector is an alias template that uses a polymorphic allocator

The elements are stored contiguously, which means that elements can be accessed not only through iterators, but also using offsets to regular pointers to elements. This means that a pointer to an element of a vector may be passed to any function that expects a pointer to an element of an array. (since C++03)

The storage of the vector is handled automatically, being expanded and contracted as needed. Vectors usually occupy more space than static arrays, because more memory is allocated to handle future growth. This way a vector does not need to reallocate each time an element is inserted, but only when the additional memory is exhausted. The total amount of allocated memory can be queried using `capacity()` function. Extra memory can be returned to the system via a call to `shrink_to_fit()`. (since C++11)
### Element access

| at          | access specified element with bounds checking (public member function) |
| operator[] | access specified element (public member function) |
| front       | access the first element (public member function) |
| back        | access the last element (public member function) |
| data(C++11) | direct access to the underlying array (public member function) |

### Capacity

| empty     | checks whether the container is empty (public member function) |
| size      | returns the number of elements (public member function) |
| max_size  | returns the maximum possible number of elements (public member function) |
| reserve   | reserves storage (public member function) |
| capacity  | returns the number of elements that can be held in currently allocated storage (public member function) |
| shrink_to_fit(C++11) | reduces memory usage by freeing unused memory (public member function) |

### Modifiers

| clear     | clears the contents (public member function) |
| insert    | inserts elements (public member function) |
| emplace(C++11) | constructs element in-place (public member function) |
| erase     | erases elements (public member function) |
| push_back | adds an element to the end (public member function) |
| emplace_back(C++11) | constructs an element in-place at the end (public member function) |
| pop_back  | removes the last element (public member function) |
| resize    | changes the number of elements stored (public member function) |
| swap      | swaps the contents (public member function) |
Stack ADT
Queue ADT
#pragma once

#include <vector>

template <typename T>
class Stack {
public:
    void push(const T & t);
    T pop();
    bool isEmpty();
private:
    std::vector<T> list_;}

#include "Stack.hpp"
template <typename T>
void Stack<T>::push(const T & d) {
    list_.push_back(d);
}

template <typename T>
T Stack<T>::pop() {
    T data = list_.back();
    list_.pop_back();
    return data;
}
Implications of Design

1. ```cpp
class ListNode {
public:
    T & data;
    ListNode * next;
    ...
}
```  

2. ```cpp
class ListNode {
public:
    T * data;  ...
}
```  

3. ```cpp
class ListNode {
public:
    T data;  ...
}
```
## Implications of Design

<table>
<thead>
<tr>
<th></th>
<th>Storage by Reference</th>
<th>Storage by Pointer</th>
<th>Storage by Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Who manages the lifecycle of the data?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Is it possible for the data structure to store NULL?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If the data is manipulated by user code while in our data structure, is the change reflected in our data structure?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Is it possible to store literals?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speed</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data Lifecycle

Storage by reference:

1. Sphere s;
2. myStack.push(s);

Storage by pointer:

1. Sphere s;
2. myStack.push(&s);

Storage by value:

1. Sphere s;
2. myStack.push(s);
Possible to store NULL?

Storage by reference:
```cpp
class ListNode {
    public:
        T & data;
        ListNode * next;
        ListNode(T & data) : data(data), next(NULL) { }
    }
```

Storage by pointer:
```cpp
T ** arr;
```

Storage by value:
```cpp
T * arr;
```
Data Modifications

```java
1  Sphere s(1);
2  myStack.push(s);
3
4  s.setRadius(42);
5
6  Sphere r = myStack.pop();
7  // What is r’s radius?
```
Speed