Reference: Big C++.

Exercises 12:

Exercise R12.1.

Insertion: n + 1 legal position: If I have 1 element, I can insert before the begin and before the end.

Erase: n legal position.

Exercise R12.2.

Assertion happens.

Exercise R12.3.

void print\_list(std::list<int> num)

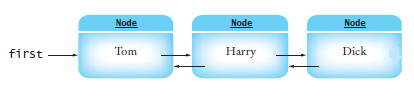
{

std::list<int>::iterator i;

for (i = num.begin(); i != num.end(); i++)

std::cout << \*i;

}

Exercise R12.4.

Exercise R12.5.

Dick Tom Harry

Exercise R12.7.

Lists are more efficient with inserting and deleting elements if the location was known before. But they don’t support random access as vectors do.

Exercise R12.8.

Vectors seem a good solution, because traversing the data in a list costs more than traversing the data in a vector. And it seems that insertion doesn’t happen frequently.

Exercise R12.9.

First appointment object is big enough and it will be inefficient to store it in a vector. Second, traversing appointment array will not happen frequently, but adding/removing appointments will occur so frequently. So, a list will be a good solution.

Exercise R12.10.

A Queue because Queues support FIFO.

If stack is used, cards will be taken from the top if the deck and returned to the top also.

Exercise R12.11.

Because if n is two large, its big-Oh will still be O(n). Unlike the access in vectors, it’s always O(1).

Exercise R12.12.

In lists, we just adjust some pointers and the node is added. But in vectors, we need to increase the size of the vector by one, then copy all the elements before/after the insertion point. This costs O(n).

Exercise R12.13.

Easy. It’s all about Capacity and size.

Exercise R12.14.

This will lead to many more reallocations -hence copy all the elements and delete the old vector. This will cost O(n) and not amortized constant time.

Exercise R12.15.

Polish Notation

1. + 3 4
2. + \* 1 2 3
3. \* 1 + 2 3
4. \* - 2 4 + 3 4
5. + + + 1 2 3 4

RPN

1. 3 4 +
2. 1 2 \* 3 +
3. 1 2 3 + \*
4. 2 4 – 3 4 + \*
5. 1 2 + 3 4 +

Exercise R12.16.

The strings will be printed through “A” to “Z”

Exercise R12.17.

It’s always O(1) because the location of the last element is always known i.e. end().

Exercise R12.18.

It will be O(1) if it doesn’t require reallocation. And O(1)+ if there’s reallocations.

Exercise R12.19.

O(1) because the begin and end locations of a list are known.

Exercise R12.20.

O(n) when popping because we will need to remove all elements and move them by one element.

O(1)+ when pushing.