ArrayList

August 8, 2020

1 Array-based List

https://opendsa-server.cs.vt.edu/ODSA/Books/CS2/html/ListADT.html

1.1 Table of Contents

- **Section ??**
- **Section ??**
- **Section ??**

1.2 List

- C++ STL provides several data structures/containers; often we need to design our own
- we'll build **list** Abstract Data Type (ADT) in this chapter
- **list**: finite, sequence of data items/elements
- some terminologies and definitions:
- a list is said to be **empty** when it contains no elements
- number of elements currently stored is called the **length**
- the beginning of the list is called **head**
- the end of the list is called **tail**

1.3 Defining List ADT

- how do we store elements? C-Array is one of the easier options!
- what basic operations do we want our list to support?
- some intuition and experience in using STL container says, we want our list to be able to:
  1. add more elements
  2. delete elements
  3. clear elements
  4. access elements
  5. know the length and many more…
- operations similar to STL Vector
- Tricky Operations:
  - insert element in the middle
  - remove element from the middle

[1]:
```
#pragma once

#include <iostream>
```

```
#include <cassert>
#include <stdexcept>

using namespace std;
```

[2]:
```
/*
Class interface and definition for dynamic array-based ListType template.

Class template definition can't be in separate file because
the compiler needs to know all the implementation details of the class
so it can instantiate different versions of the code, depending on the
actual types provided for each template parameters.

Remember that a template doesn't represent code directly, but a
template for several versions of that code.
*/

template <class T>
class ListType
{
private:
    T *list;
    int length;
    int maxSize;
public:
    ListType(size_t max = 10000); // constructor
    ListType(const ListType<T> &other); // copy constructor
    ~ListType(); // destructor

    // **** LIST META-DATA AND CAPACITY INFO ****
    // return true if list is empty
    bool isEmpty() const;
    // return true if list is full
    bool isFull() const;
    // get the length/size of the list
    int listSize() const;
    // get the max list size
    int listMaxSize() const;

    // **** ELEMENT ACCESS ****
    // retrieve the element at given index
    T at(size_t index) const;
    // access the last/back element
    T back();
    // access the first element
    T front();
    // overload operator[] as member function
```

```
    T operator[](int index) const;

    // **** MODIFIERS ***
    // insert at the end of the list
    void pushBack(const T &item);
    // delete the last element
    void popBack();
    // insert given item at the given index
    void insertAt(size_t index, const T &item);
    // deletes all the elements in the list and resets the list
    void clear();
    // replace element at index with new item
    void replaceAt(size_t index, const T &item);
    // remove the element at given index
    void removeAt(size_t index);

    // do a linear search on given searchItem and return index if found -1␣
 ↪otherwise
    int search(const T &searchItem);
    // find the item in the container and remove it
    void remove(const T &item);
};

template <class T>
ListType<T>::ListType(size_t max) {
    this->maxSize = max;
    this->length = 0;
    this->list = new T[maxSize]; // dynamic array!
}

template <class T>
ListType<T>::ListType(const ListType<T> &other) {
    this->length = other.length;
    this->maxSize = other.maxSize;
    this->list = new T[maxSize];
    for (int i = 0; i<other.length; i++)
        this->list[i] = other.list[i];
}

template <class T>
ListType<T>::~ListType() {
    delete[] list;
}

// return true if list is empty
template <class T>
bool ListType<T>::isEmpty() const {
```

```
    return (length == 0);
}

// return true if list is full
template <class T>
bool ListType<T>::isFull() const {
    return (length == maxSize);
}

// return size/length of the list
template <class T>
int ListType<T>::listSize() const {
    return this->length;
}

// return the max size of list
template <class T>
int ListType<T>::listMaxSize() const {
    return this->maxSize;
}

// retrieve the element at given index
template <class T>
T ListType<T>::at(size_t index) const {
    if (index < 0 || index >= length)
       throw out_of_range("Index out of range.");

    return list[index];
}

template<class T>
T ListType<T>::back() {
    // doesn't check if the list is empty!
    return list[length-1];
}

template<class T>
T ListType<T>::front() {
    // doesn't check if the list is empty!
    return list[0];
}

//return the reference to the value at given index
template <class T>
T ListType<T>::operator[](int index) const {
    if (index < 0 || index >= length)
        throw out_of_range("Index out of range.");
```

```
    return list[index];
}

// insert at the end of the list
template <class T>
void ListType<T>::pushBack(const T &item) {
    if (isFull())
        throw overflow_error("List is full.");
    else {
        list[length] = item;
        length++;
    }
}

// delete the last element
template <class T>
void ListType<T>::popBack() {
    removeAt(length-1);
}

// insert item at the index
// Exception: out_of_range thrown when list it full or index is out of bounds
template <class T>
void ListType<T>::insertAt(size_t index, const T &item) {
    if (index < 0 || index >= length)
       throw out_of_range("Index out of range.");
    else if (isFull()) {
        throw overflow_error("List is full.");
    }
    else {
        // move the elements down from the index
        // starting from the end
        for (int i = length; i > index; i--)
            list[i] = list[i - 1];
        list[index] = item; // insert the item at the specified index
        length++; // increment the length
    }
}

// remove the element at given index
template <class T>
void ListType<T>::removeAt(size_t index) {
    if (index < 0 || index >= length)
       throw out_of_range("Index out of range.");
    else {
        // move elements up one position one at a time
        for (int i = index; i < length - 1; i++)
```

```
            list[i] = list[i + 1];
        length--; // decrease list length by 1
    }
}

// deletes all the elements in the list and resets the list
template <class T>
void ListType<T>::clear() {
    length = 0;
}

// replace element at index with new item
template <class T>
void ListType<T>::replaceAt(size_t index, const T &item) {
    if (index < 0 || index >= length)
        throw out_of_range("Index out of range.");
    else
        list[index] = item;
}

// do a linear search on given searchItem and return index if found, -1␣
 ↪otherwise
template <class T>
int ListType<T>::search(const T &searchItem) {
    int index = 0;
    bool found = false;
    while (index < length && !found)
    {
        if (list[index] == searchItem)
            found = true;
        else
            index++;
    }
    if (found)
        return index;
    else
        return -1;
}

// find the item in the container and remove it
template <class T>
void ListType<T>::remove(const T &item)
{
    int index = -1;
    if (isEmpty())
        throw underflow_error("List is empty.");
    else
```

```
    {
        index = seqSearch(item);
        if (index != -1)
            removeAt(index);
    }
}
```

[3]:
```
// print all the elements in the list
// overload operator<<
template<class T>
ostream& operator<<(ostream& out, const ListType<T>& alist){
    out << "max size = " << alist.listMaxSize() << endl;
    out << "length = " << alist.listSize() << endl;
    out << "list contents: " << endl;
    for (int i = 0; i<alist.listSize(); i++)
        out << alist[i] << " ";
    out << endl;
    return out;
}
```

1.4 Using ListType ADT

[4]:
```
// Test ListType
ListType<int> ilist(100);
```

[5]:
```
ilist.pushBack(10);
cout << ilist;
```

```
max size = 100
length = 1
list contents:
10
```

[6]:
```
ilist.insertAt(0, 20);
cout << ilist;
```

```
max size = 100
length = 2
list contents:
20 10
```

[7]:
```
ilist.clear();
cout << ilist;
```

```
max size = 100
length = 0
```

```
list contents:
```

[8]:
```
ilist.replaceAt(0, 200);
cout << ilist;
```

```
        Error:
```

[9]:
```
int i;
```

[10]:
```
i = ilist.search(10);
if (i < 0)
    cout << "element not found...";
else
    cout << "element found at index: " << i << endl;
```

```
element not found…
```

1.4.1 Exercise

- In an array-based list, the successive elements in the list:
  1. Need not occupy contiguous space in memory
  – Must occupy contiguous space in memory
  – None of these
  – Must not occupy contigious space in memory

[ ]:
```

```