5.1 arrayList & linked list

Tuesday, September 27, 2022

TODAY: two implementations of the List ADT

LIST: an ordered sequence of elements all of the same type

Overall idea: want a more user-friendly data type so we don't have to worry about out-of-bounds indexing, memory management, etc. when using lists.

declaration of List ADT:

see this file:
/home/fontes/public/cs35/week5/ArrayList/list.h

```
template <typename T>
class List {
  public:

    virtual ~List; // destructor
    virtual int getSize() = 0;
    virtual bool isEmpty() = 0;
    virtual T get(int i) = 0; // return the element at index i
    virtual T getFirst() = 0;
    virtual T getLast() = 0;
    virtual void insertFirst(T value) = 0;
    virtual void insertLast(T value) = 0;
    virtual T removeFirst() = 0;
    virtual T removeLast() = 0;

};
```

We'll think about each list having
- first (head)
- last (tail)
- elements

We will implement List in two ways:

ArrayList LinkedList

Assuming we will have an implementation, let's trace some code:

```
LinkedList<int> numbers;
numbers.insertFirst(2);
numbers.insertFirst(6);
numbers.insertFirst(3);
```

NOT A STACK DIAGRAM

numbers
head tail

head → 2 ← tail

head → 6 → 2 ← tail

head → 3 → 6 → 2 ← tail

ArrayList interface & implementation details:

see this file: /home/fontes/public/cs35/week5/ArrayList/arrayList.h

```
1  #pragma once
2
3  #include "list.h"
4
5  template <typename T>
6  class ArrayList : public List<T> {
7   public:
8    ArrayList();
9    ArrayList(int capacity);
10   ~ArrayList();
11
12   void checkInvariants();
13   int getSize();
14   bool isEmpty();
15   T getFirst();
16   T getLast();
17   T get(int i);
18   void insertFirst(T value);
19   void insertLast(T value);
20   T removeFirst();
21   T removeLast();
22
23  private:
24   T* array;
25   int size;
26   int capacity;
27   void ensureCapacity();
28 };
29
30 #include "arrayList-inl.h"
```

} two constructors

← helpful for users

← helper for later implementation details

ArrayList constructors & destructor

see these implementations in:
/home/fontes/public/cs35/week3/ArrayList-inl.h

```
10 template <typename T>                          O(1)
11 ArrayList<T>::ArrayList() {
12   // choose 10 as a default size
13   this->array = new T[10];
14   this->size = 0;
15   this->capacity = 10;
16 }
17
18 template <typename T>                          O(1)
19 ArrayList<T>::ArrayList(int capacity) {
20   this->array = new T[capacity];
21   this->size = 0;
22   this->capacity = capacity;
23 }
24
25 template <typename T>                          O(1)
26 ArrayList<T>::~ArrayList() {
27   // delete dynamic memory that was used by the ArrayList
28   delete[] this->array;
29   this->array = nullptr;
30 }
```

two constructors set up the array & capacity differently

this line is a safety/sanity check

ArrayList getSize & isEmpty & get

```
42 template <typename T>
43 int ArrayList<T>::getSize() {                  O(1)
44   return this->size;
45 }
46
47 template <typename T>
48 bool ArrayList<T>::isEmpty() {                 O(1)
49   return (this->size==0);
50 }
51
72 template <typename T>
73 T ArrayList<T>::get(int i) {
74   if (i >= 0 && i < this->size) {
75     return this->array[i];
76   }
77   else{
78     throw runtime_error("ArrayList::get index out of bounds");
79   }
80 }
```

get is also O(1)

ArrayList check invariants

idea: we want to make sure certain properties ("invariants") are always true

For an ArrayList with a specific size and specific capacity,

size ≥ 0 } these conditions should

size ≥ 0
size ≤ capacity
} these conditions should be TRUE

```
template <typename T>
void ArrayList<T>::checkInvariants() {
  if (this->size < 0) {
    throw runtime_error("ArrayList's size must be non-negative");
  }
  if (this->size > this->capacity) {
    throw runtime_error("ArrayList's size must be <= its capacity");
  }
}
```

O(1)

ArrayList inserting an element

```
template <typename T>
void ArrayList<T>::insertFirst(T value) {
  ensureCapacity();
  for(int i=this->size - 1; i>=0 ; i--) {
    this->array[i+1] = this->array[i];
  }
  this->array[0] = value;
  this->size++;
}

template <typename T>
void ArrayList<T>::insertLast(T value) {
  ensureCapacity();
  this->array[this->size] = value;
  this->size++;
}
```

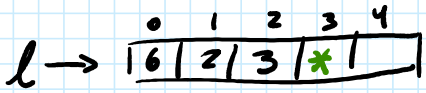
O(n)

helper function to make sure there is available space in the array

O(n) worst case

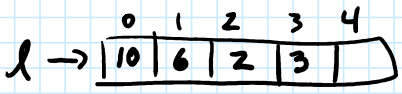
O(n) worst case

} O(1)



size = 3
capacity = 5

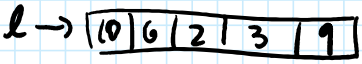
l → insertFirst(10);



i = 3 2 1 0 -1

size = 4
capacity = 5

l → insertLast(9);



size = 5
capacity = 5

ArrayList ensureCapacity

idea: make sure there is some available space in the array

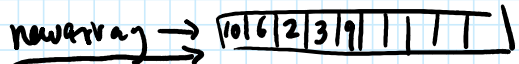
```
template <typename T>
void ArrayList<T>::ensureCapacity() {
  if (this->size == this->capacity) {
    int newCapacity = 2 * this->capacity;
    T* newarray = new T[newCapacity];
    for (int i=0; i<this->size; i++) {
      newarray[i] = this->array[i];
    }
    delete[] this->array;
    this->array = newarray;
    this->capacity = newCapacity;
  }
}
```

O(n)

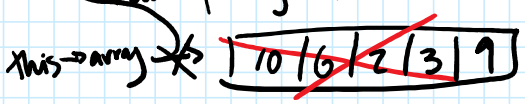
this means the array is full

like this

l → ensureCapacity();



newCapacity = 10



this->capacity = 10

ArrayList removing an item

ArrayList removing an item

```
99 template <typename T>
100 T ArrayList<T>::removeFirst() {
101   if (this->size != 0){
102     // store the first element in a temporary variable
103     T temp = this->array[0];
104     // move every other element of the array one to the left
105     for (int i=1; i<this->size ; i++){
106       this->array[i-1] = this->array[i];
107     }
108     this->size--; // decrease the size
109     return temp;  // return the element that was removed
110   }
111   else {
112     throw runtime_error("ArrayList::removeFirst called on empty list.");
113   }
114 }
115
116 template <typename T>
117 T ArrayList<T>::removeLast() {
118   if (this->size != 0) {
119     // just reduce size by one and 'forget' that an element was there
120     this->size--;
121     return this->array[this->size];
122   }
123   else {
124     throw runtime_error("ArrayList::removeLast called on empty list.");
125   }
126 }
```

O(n)

O(1)

l -> removeFirst();

l -> | 10 | 6 | 2 | 3 | 9 | | | | |

temp = 10

returns 10

l -> | 6 | 2 | 3 | 9 | | | | | |

l -> removeLast();

returns 9

size-1 = 2 , size = 3

l -> | 6 | 2 | 3 |

considered as empty

OPERATION	ARRAYLIST runtime	
int getSize( )	O(1)	
bool isEmpty( )	O(1)	
T get(i)	O(1)	
T getFirst( )	O(1)	
T getLast( )	O(1)	
void insertFirst(T)	O(n)	← expensive
void insertLast(T)	O(n) worst case O(1) amortized	← sort of expensive, in worst case
T removeFirst( )	O(n)	← expensive
T removeLast( )	O(1)	

Next time: can we do better with a different implementation of the List ADT?