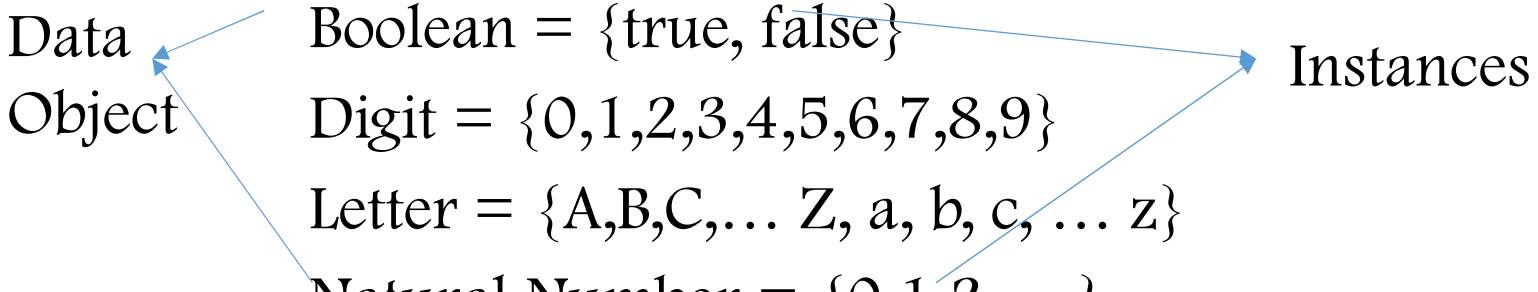
Linear Lists – Array Representation
Chapter - 5

Data Object

A data object is a set of instances or values.

Examples:



instances may or may not be related

myDataObject = {apple, chair, 2, 5.2, red, green, Jack}

Data Structure

A data structure is a data object together with the relationships that exist among the instances and among the individual elements that compare an instance.

Linear List (or ordered list) is an ordered collection of element. Each instance is of the form $(e_0, e_1 \ldots e_{n-1})$

where n is a finite natural number,

e_i are the elements of the list,

n is the list length

Linear Lists

A linear list may be specified as an Abstract Data Type (ADT) in which we provide a specification of the instances as well as of the operations that are to be performed

$L = (e_0, e_1, e_2, e_3, \ldots, e_{n-1})$

relationships

e_0 is the zero'th (or front) element

e_{n-1} is the last element

e_i immediately precedes e_{i+1}

Linear List Examples/Instances

Students in Course = (Jack, Jill, Abe, Henry, Mary, …, Judy)

Exams in Course =(exam1, exam2, exam3)

Days of Week = (S, M, T, W, Th, F, Sa)

Months = (Jan, Feb, Mar, Apr, …, Nov, Dec)

Linear List Abstract Data Type

AbstractDataType LinearList

{

instances

ordered finite collections of zero or more elements

operations

IsEmpty(): return true iff the list is empty, false otherwise

Length(): return the list size (i.e., number of elements in the list)

Retrieve(index): return the indexth element of the list

IndexO f(x): return the index of the first occurrence of x in the list, return -1 if x is not in the list

Delete(index): remove and return the indexth element, elements with higher index have their index reduced by 1

Insert(theIndex, x): insert x as the indexth element, elements with theIndex >= index have their index increased by 1

}

Linear List Operations
~Length() or Size()

determine number of elements in list

L = (a,b,c,d,e)

length = 5

Linear List Operations—Retrieve(theIndex) or Get(theIndex)

retrieve element with given index

L = (a,b,c,d,e)

Retrieve(0) = a

Retrieve(2) = c

Retrieve(4) = e

Retrieve(-1) = error

Retrieve(9) = error

Linear List Operations—IndexOf(theElement)

determine the index of an element

L = (a,b,d,b,a)

IndexOf(d) = 2

IndexOf(a) = 0

IndexOf(z) = -1

Linear List Operations—Delete(theIndex) Or Erase(theindex)

delete and return element with given index

L = (a,b,c,d,e,f,g)

Delete(2) returns c

and L becomes (a,b,d,e,f,g)

index of d,e,f, and g decrease by 1

Linear List Operations—Delete(theIndex)

delete and return element with given index

L = (a,b,c,d,e,f,g)

Delete(-1) => error

Delete(20) => error

Linear List Operations—Insert(theIndex, theElement)

insert an element so that the new element has a specified index

L = (a,b,c,d,e,f,g)

Insert(0,h) => L = (h,a,b,c,d,e,f,g)

index of a,b,c,d,e,f, and g increase by 1

Linear List Operations—Insert(theIndex, theElement)

L = (a,b,c,d,e,f,g)

Insert(2,h) => L = (a,b,h,c,d,e,f,g)

index of c,d,e,f, and g increase by 1

Insert(10,h) => error

Insert(-6,h) => error

Linear List Operations—Output()

L = (a,b,c,d,e,f,g)

Output the list elements from left to right

a, b, c, d, e, f, g

Let $L = (a, b, c, d)$ be a linear list. What is the result of each of the following operations?

(a) $empty()$

(b) $size()$

(c) $get(0)$, $get(2)$, $get(6)$, $get(-3)$

(d) $indexOf(a)$, $indexOf(c)$, $indexOf(q)$

(e) $erase(0)$, $erase(2)$, $erase(3)$

(f) $insert(0, e)$, $insert(2, f)$, $insert(3, g)$, $insert(4, h)$, $insert(6, h)$, $insert(-3, h)$

(a) false

(b) 4

(c) a, c, there is no element whose index is 6 (error), there is no element whose index is -3 (error).

(d)0, 2, -1.

(e) Assume that each operation is done on the initial list.
The list becomes (b,c,d) and a is returned.
The list becomes (a,b,d) and c is returned.
The list becomes (a,b,c) and d is returned.

(f) Assume that each operation is done on the initial list.
(e,a,b,c,d), (a,b,f,c,d),(a,b,c,f,d)., (a,b,c,d,f), the remaining two are invalid operations.

Data Structure Specification

- C++ supports two types of classes – **abstract** and **concrete.**

- An **abstract** class is a class that contains a member function for which no implementation has been specified. Such a function is called **pure virtual function**

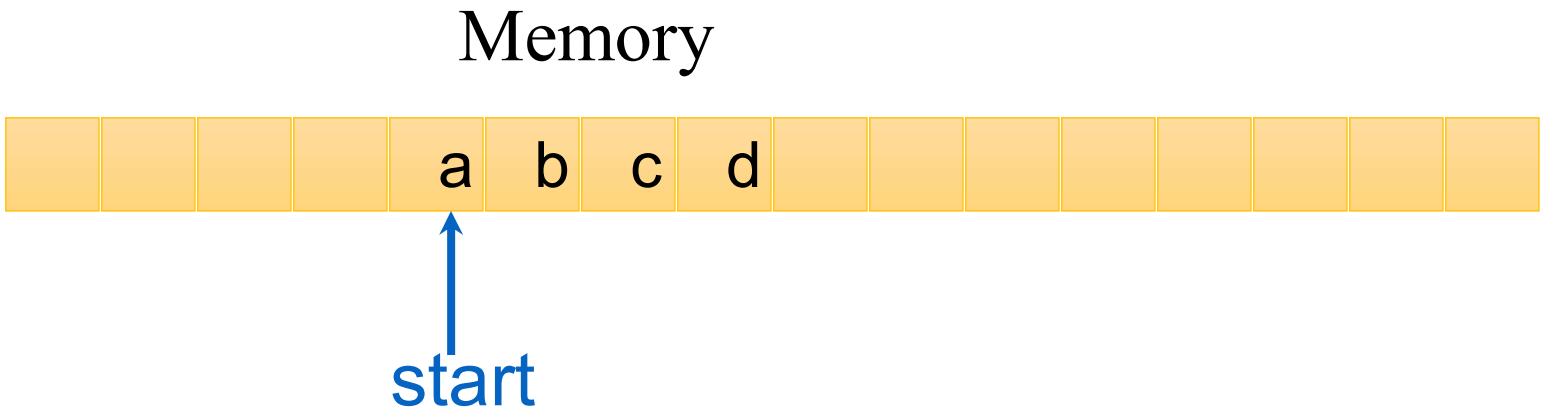
```
virtual int myPureVirtualFunction(int x) = 0;
```

- A **concrete** class contains no pure virtual function. Only concrete class may be instantiated.

Linear List As A C++ Class

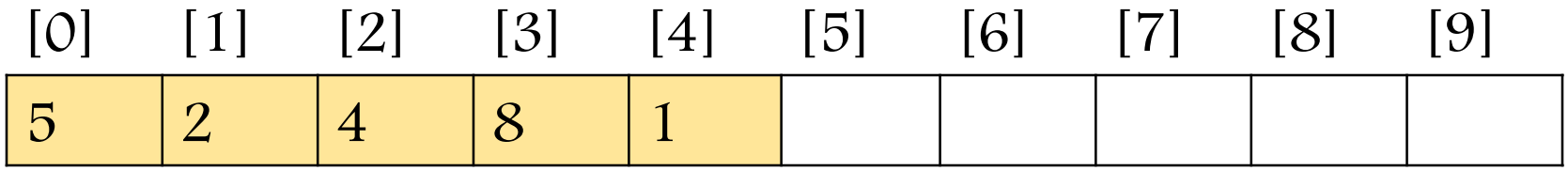
```
class LinearListOfIntegers
{
   bool IsEmpty() const;
   int length() const;
   int Retrieve(int index) const;
   int IndexOf(int theElement) const;
   int Delete(int index);
   void Insert(int index, int theElement);
}
```

1D Array Representation In C++

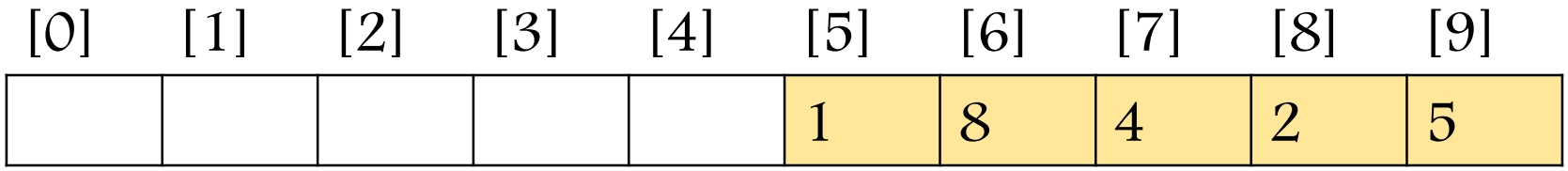


- 1-dimensional array x = [a, b, c, d]
- map into contiguous memory locations
- location(x[i]) = start + i

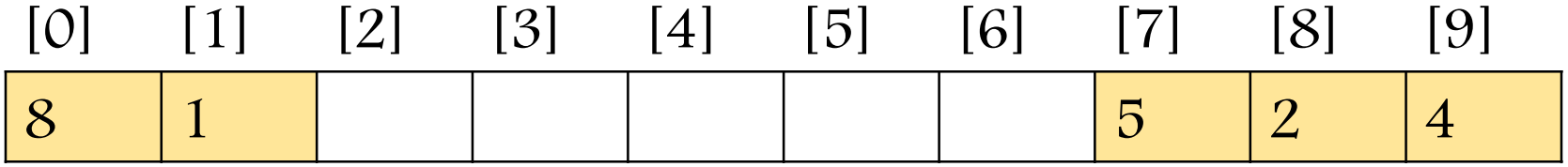
Different ways to map the elements into 1D-array



(a) Location (i) = i



(b) Location (i) = 9-i = arraylength-1-i



(c) Location (i) = (7+i)%10 = (location(0) + i) %arraylength

Changing the length of 1D-array

- Takes $\theta(1)$ time to create an array of length m.
- $\Theta(n)$ time is needed to copy elements from source array into destination array
- Program complexity is $O(n)$.

```
template<class T>
void changeLength1D(T*& a, int oldLength, int newLength)
{
   if (newLength < 0)
      throw illegalParameterValue("new length must be >= 0");

   T* temp = new T[newLength];              // new array
   int number = min(oldLength, newLength);  // number to copy
   copy(a, a + number, temp);
   delete [] a;                             // deallocate old memory
   a = temp;
}
```

Class Definition of Array

```
template<class T>
class arrayList : public linearList<T>
{
   public:
      // constructor, copy constructor and destructor
      arrayList(int initialCapacity = 10);
      arrayList(const arrayList<T>&);
      ~arrayList() {delete [] element;}

      // ADT methods
      bool empty() const {return listSize == 0;}
      int size() const {return listSize;}
      T& get(int theIndex) const;
      int indexOf(const T& theElement) const;
      void erase(int theIndex);
      void insert(int theIndex, const T& theElement);
      void output(ostream& out) const;

      // additional method
      int capacity() const {return arrayLength;}

   protected:
      void checkIndex(int theIndex) const;
            // throw illegalIndex if theIndex invalid
      T* element;            // 1D array to hold list elements
      int arrayLength;       // capacity of the 1D array
      int listSize;          // number of elements in list
};
```

Constructors

```
template<class T>
arrayList<T>::arrayList(int initialCapacity)
{// Constructor.
   if (initialCapacity < 1)
   {ostringstream s;
    s << "Initial capacity = " << initialCapacity << " Must be > 0";
    throw illegalParameterValue(s.str());
   }
   arrayLength = initialCapacity;
   element = new T[arrayLength];
   listSize = 0;
}

template<class T>
arrayList<T>::arrayList(const arrayList<T>& theList)
{// Copy constructor.
   arrayLength = theList.arrayLength;
   listSize = theList.listSize;
   element = new T[arrayLength];
   copy(theList.element, theList.element + listSize, element);
}
```

Instantiating arrayList

```
// create two linear lists with initial capacity 100
linearList *x = (linearList) new arrayList<int>(100);
arrayList<double> y(100);

// create a linear list with the default initial capacity
arrayList<char> z;

// create a linear list that is a copy of the list y
arrayList<double> w(y);
```

Elementary Methods

```
template<class T>
void arrayList<T>::checkIndex(int theIndex) const
{// Verify that theIndex is between 0 and listSize - 1.
   if (theIndex < 0 || theIndex >= listSize)
   {ostringstream s;
    s << "index = " << theIndex << " size = " << listSize;
    throw illegalIndex(s.str());
   }

}

template<class T>
T& arrayList<T>::get(int theIndex) const
{// Return element whose index is theIndex.
 // Throw illegalIndex exception if no such element.
   checkIndex(theIndex);
   return element[theIndex];
}

template<class T>
int arrayList<T>::indexOf(const T& theElement) const
{// Return index of first occurrence of theElement.
 // Return -1 if theElement not in list.

   // search for theElement
   int theIndex = (int) (find(element, element + listSize, theElement)
                         - element);

   // check if theElement was found
   if (theIndex == listSize)
     // not found
     return -1;
   else return theIndex;
 }
```

Remove an element

```
template<class T>
void arrayList<T>::erase(int theIndex)
{// Delete the element whose index is theIndex.
 // Throw illegalIndex exception if no such element.
   checkIndex(theIndex);

   // valid index, shift elements with higher index
   copy(element + theIndex + 1, element + listSize,
                                element + theIndex);

   element[--listSize].~T(); // invoke destructor
}
```

Insert an element

```
template<class T>
void arrayList<T>::insert(int theIndex, const T& theElement)
{// Insert theElement so that its index is theIndex.
   if (theIndex < 0 || theIndex > listSize)
   {// invalid index
      ostringstream s;
      s << "index = " << theIndex << " size = " << listSize;
      throw illegalIndex(s.str());
   }

   // valid index, make sure we have space
   if (listSize == arrayLength)
      {// no space, double capacity
         changeLength1D(element, arrayLength, 2 * arrayLength);
         arrayLength *= 2;
      }

   // shift elements right one position
   copy_backward(element + theIndex, element + listSize,
                 element + listSize + 1);

   element[theIndex] = theElement;

   listSize++;
}
```

Array linear list using vector

```
template<class T>
class vectorList : public linearList<T>
{
   public:
      // constructor, copy constructor and destructor
      vectorList(int initialCapacity = 10);
      vectorList(const vectorList<T>&);
      ~vectorList() {delete element;}

      // ADT methods
      bool empty() const {return element->empty();}
      int size() const {return (int) element->size();}
      T& get(int theIndex) const;
      int indexOf(const T& theElement) const;
      void erase(int theIndex);
      void insert(int theIndex, const T& theElement);
      void output(ostream& out) const;

      // additional method
      int capacity() const {return (int) element->capacity();}

      // iterators to start and end of list
      typedef typename vector<T>::iterator iterator;
      iterator begin() {return element->begin();}
      iterator end() {return element->end();}

   protected:  // additional members of vectorList
      void checkIndex(int theIndex) const;
      vector<T>* element;     // vector to hold list elements
};
```

Iterators in C++

- An iterator is a pointer to an element of an object.
- In for loop, y , which is a pointer, is incremented every time

```
int main()
{
   int x[3] = {0, 1, 2};

   // use a pointer y to iterate through the array x
   for (int* y = x; y != x + 3; y++)
      cout << *y << "  ";
   cout << endl;
   return 0;
}
```

Exercise

- Write a function ChangeLength2D to change the length of a two dimensional array. Allow change in both directions of the array

```
template<class T>
void changeLength2D(T**& a, int oldRows, int copyRows,
                    int copyColumns, int newRows, int newColumns)
{// Resize the two-dimensional array a that has oldRows number of rows.
 // The dimensions of the resized array are newRows x newColumns.
 // Copy the top left oldRows x newColumns sub array into the resized array.
   // make sure new dimensions are adequate
   if (copyRows > newRows || copyColumns > newColumns)
      throw illegalParameterValue("new dimensions are too small");

   T** temp = new T*[newRows];              // array for rows
   // create row arrays for temp
   for (int i = 0; i < newRows; i++)
      temp[i] = new T[newColumns];

   // copy from old space to new space, delete old space
   for (int i = 0; i < copyRows; i++)
   {
      copy(a[i], a[i] + copyColumns, temp[i]);
      delete [] a[i];                       // deallocate old memory
   }

   // delete remaining rows of a
   for (int i = copyRows; i < oldRows; i++)
     delete [] a[i];

   delete [] a;
   a = temp;
}
```

- Overload the operator [] so that the expression x[i] returns a reference to the ith element of the list. If the list doesn't have an ith element, an illegal Index exception is to be thrown. The statements x[i] = y and y = x[i] should work as expected

```
template<class T>
class arrayListWithOverloadIndex : public arrayList<T>
{
   public:
      // constructor and destructor
      arrayListWithOverloadIndex(int initialCapacity = 10)
           : arrayList<T> (initialCapacity) {}

      T& operator[](int);
};

template<class T>
T& arrayListWithOverloadIndex<T>::operator[](int theIndex)
{// Return a reference to position theIndex of the list.

    checkIndex(theIndex);
    return element[theIndex];
}
```

Write the method `arrayList<T>::leftShift(i)` that shifts the list elements left by `i` positions. If `x` = [0, 1, 2, 3, 4], then `x.leftShift(2)` results in `x` = [2, 3, 4].

What is the time complexity of your method?

```
template<class T>
class arrayListWithLeftShift : public arrayList<T>
{
   public:
      // constructor and destructor
      arrayListWithLeftShift(int initialCapacity = 10)
           : arrayList<T> (initialCapacity) {}

      void leftShift(int theAmount);
};

template<class T>
void arrayListWithLeftShift<T>::leftShift(int theAmount)
{// Left shift by i elements.


   if (theAmount <= 0)
      return;
      // could throw exception when i < 0 or do a right shift by i

   int newSize = 0;
   if (theAmount < listSize)
   {// list is not empty after the shift
      newSize = listSize - theAmount;

      // left shift elements with higher index >= i
      copy(element + theAmount, element + listSize, element);
   }

   // destroy uneeded elements
   for (int i = newSize; i < listSize; i++)
      element[i].~T();

   listSize = newSize;
}
```

The complexity of the method is O(listSize).