Chapter 3 ADT Lists

Goals

Look at classical ADTs: lists, queues and stacks

- How an ADT is defined: logical structures of the data and operations on the data
- How to use it after it is defined, even before it is implemented
- How to implement the ADT, including designing the physical structures and implementing the operations
- Analyze performances of the operations.

Abstract Data Types

- A data type consists of a collection of values together with a set of basic operations on these values
- A data type is an abstract data type if the programmers who use the type do not have access to the details of how the values and operations are implemented.
- All pre-defined types such as int, double, string … are abstract data types
- An abstract data type is abstract in the sense that the implementation is 'abstract'

ADTs in C++

An Abstract Data Type is implemented as a class: data become *private members* and operations are represented by *public methods*.

Make all the member variables private

→ private data (implementation details)

Make member functions public

→ public interface

Abstract Data Type

- Using *Encapsulation and Information Hiding*: data members can only be accessed by methods, and realize the separation of user interface with implementation details.
- Advantages: easy and efficient programming: as long as the interface stays stable, the implementations can be changed without affecting client code.

Rational Review

- Rational number
  - Ratio of two integers: `a/b`
    - Numerator over the denominator
- Standard operations:

$$\frac{a}{b} \pm \frac{c}{d} = \frac{ad \pm bc}{bd}$$

$$\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$$

$$\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$$

Rational Representation

- Represent a numerator and denominator with two `int` data members
  - `Numerator` and `Denominator`
  - Data members private (information hiding)
- Public arithmetic member functions
  - Rational addition, subtraction, multiplication, division
- Public relational member functions
  - Equality and less than comparisons

User's View of Rational

```
class Rational {
 public:
    Rational();
    Rational(int numer, int denom = 1);
                    // arithmetic functions
    const Rational & operator+(const Rational r);
    const Rational & operator-(const Rational r);

    ...

    void Display() const;
} ;
```

Example

```
void main(){
        Rational r(1,2);
        Rational s(1,2);
        Rational t = r + s;

        cout << "The sum of r and s: ";
        t.Display();
        t = r*s;
        cout << "The product of r and s: ";
        t.Display();

}
```

The Rational Class

```
class Rational{
   public:
           // default-value constructor
        Rational();
           // explicit-value constructor
        Rational(int numer, int denom = 1);
           // arithmetic functions
        const Rational & operator+(const Rational r);
        const Rational & operator-(const Rational r);
        const Rational & operator*(const Rational r);
        const Rational & operator%(const Rational r);

        void Display() const;

   private:   // data members
        int Numerator;
        int Denominator;
};
```

List Motivation

- A "List" is a useful structure to hold a collection of data.
  - Currently, we use arrays for lists
- Examples:

  List of 63 student marks

  int studentMarks[63];

  List of temperatures for the last year

  double temperature[365];

The List ADT

- A list is a sequence of objects of the same type: $A_0, A_1, \ldots, A_{N-1}$.
  - N is the *size* of the list. *Empty list* when N=0
  - A_{i-1} is the predecessor of A_i, or A_i is the successor of A_{i-1}, where i is the position of A_i.
- Operations:
  - Insertion of an item in some position;
  - Deletion of an item in some position;
  - Location of some item;
  - Check if it is empty;
  - Check its size;
  - Print all items in the list, etc.

线性表的ADT定义

ADT List{

数据对象: $D=\{a_i \mid a_i \in Elemset, i=1,2,\ldots,n, n>=0\}$

数据关系: $R=\{(a_{i-1}, a_i) \mid a_i \in ElemSet, i=2,\ldots,n\}$

基本操作:

initList(&L)
操作结果:构造一个空的线性表L

destroyList(&L)
初始条件: 线性表L已存在;
操作结果:消毁L

getElem(L, i,&e)
初始条件: 线性表L已存在;
操作结果:用e返回线性表L第i个位置的值.

---

listInsert(&L,i, e)

初始条件: 线性表L已存在, 1=<i<=listLength(L)+1;

操作结果: 在L的第i个位置插入e，L的长度加1.

listDelete(&L,i, &e)

初始条件: 线性表L已存在, 1=<i<=listLength(L);

操作结果: 删除L的第i个位置之元素,用e返回其值，L的长度减1.

listTraverse(L, visit())

初始条件: 线性表L已存在

操作结果: 对L的每个元素调用visit(), 如果visit()失败，则操作失败。

} **ADT** List

The List Class Interface

```
template <typename List_entry>
class List {
public:
//  methods of the List ADT
   List();//construct an empty list
   int size() const;
   bool empty() const;
   void clear();
   int find( List_entry &x) const; //find the first occurrence of x
   int insert(int position, const List_entry &x);//insert x at position
   int push_back(const List_entry &x);//put x after the last item
   int erase(int position);
   void traverse(void (*visit)(List_entry &));
}
```

Example

```
void main(){
   List<int> l; //l =()
   for (int i=1;i<10;i++){
     l.push_back(i);
   };  // l = (1,2,3,…,9);
   l.insert(0,100); // l = (100,1,2,…,9)
   l.erase(0);//l = (1,2,…,9)
   l.traverse(print);// output: 1 2 3 …  9
   l.traverse(double); l = (2,4,6,…,18)
   l.traverse(print); //output: 2  4  6 …  18
}

void print( int x){
      cout << x <<"  ";
  }
void double(int &x){
    x = 2*x;
}
```

List Implementations

- list using static array, logical relations of objects realized by their physical positions in storage.

  `int myArray[1000];`

  We have to decide (to oversize) in advance the size of the array (list)
- list using dynamic array

  ```
  int* myArray;
  int n;
  cin >> n;
  myArray = new int[n];
  ```

  We allocate an array (list) of any specified size while the program is running
- linked-list (dynamic size): logical relations realized by links.

  size = ??

  The list is dynamic. It can grow and shrink to any size.

Array Implementation

```
template <typename List_entry>
class Vector {
public: //  methods of the List ADT
   Vector();//construct an empty list
   int size() const;
   bool empty() const;
   void clear();
   List_entry & operator[](int position);
   int find( List_entry &x) const; //find the first occurrence of x
   int insert(int position, const List_entry &x);//insert x at position
   int push_back(const List_entry &x);//put x after the last item
   int erase(int position);
   void traverse(void (*visit)(List_entry &));
 private:
    List_entry *elems;
    int count; //number of items in the list
    int arraySize; // the size of the array
}
```

Anything missing?

Add:
~Vector();
Vector(const Vector&);
Const Vector& operator-(const Vector&);

If a method is not const, then you must maintain the data members.

More about traverse

- Implementation

```
void traverse(void (*visit)(List_entry &)){
    for (i=0;i<n; i++)
        (*visit)(elem[i]);
}
```

- Applications
- In STL: for_each(InputIterator first, InputIterator last, UnaryFunction f)

Time Complexity

- Insertion:
  - Push_back: O(1)
  - worst case: T(n) = O(n); average case: O(n)
- Deletion:
  - Worst case: O(n); average case: O(n)
- Traversal: depends on the action parameter.
- When to use Vector?
  - Indexable in constant time, also known as “随机存取结构”
  - Insertions and deletions can be expensive except at the end;

STL Implementation

- A template class vector, which is the class encapsulation of arrays;
- An associated iterator that is abstraction of pointers and can iterate a range of objects, through which STL generic algorithms can manipulate data in containers.
- Iterator of vectors is a model of random access iterator, which means i+n, i-n and i[n] (equivalent to *(i+n)) are valid expressions if i is an iterator and n is convertible to int.
- Use vectors whenever arrays can be used: it is safe and efficient, and it has got convenient, efficient operations.

Using vectors

- #include<vector> and provide type parameter of your vector: vector<int> v;
- For const vector, use const_iterator;
- When using the operator [], make sure the index is valid;
- Insertion, deletion and memory reallocation may make an iterator invalid. For example, insertion in the middle will invalidate all iterators following that position.
- Vector demo: insertion, deletion and traversal.
- Vector exercise: write a function that merges two ordered lists of integers into an ordered list of integers.

Linked List Implementation

- The list of objects are stored in a linked list, instead of arrays;
- Linked list allows O(1) insertions and deletions;
- The list can grow and shrink as insertions and deletions are done without worrying about the capacity.

The Linked List Implementation

```
template <typename T>
class List {
 public:
//  methods of the List ADT
   List();//construct an empty list
   ~List();
   List(const List &l)
   const List & operator=(const List &l);
   int size() const;
   bool empty() const;
   void clear();
   Node* find( T &x) const; //find the first occurrence of x
   Node* insert(Node* position, const T &x);//insert x at position
   void push_back(const T &x);//put x after the last item
   void push_front(const T &x);
   T & front();
   T & back();
   int erase(Node* position);
   void traverse(void (*visit)(T &));
```

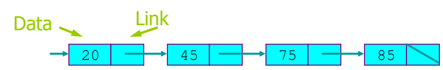
How to construct the list (12,23,34,45,56)?
How to print the list?

```
private:

   Node *head;
   Node *tail;
   int theSize;
};

struct Node {
      T data;
       Node *next;
      Node(const T & x, Node * p =NULL):data(x),next(p){ }
   };
```

Linked Lists: Basic Idea

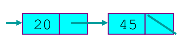
- A linked list is an *sequence* of data
- Each element of the linked list has
  - Some **data**
  - A **link** to the next element
- The link is used to chain the data

  Example: A linked list of integers:

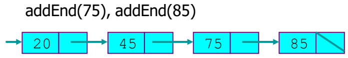


Linked Lists: Basic Ideas

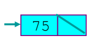
- The list can grow and shrink



addEnd(75), addEnd(85)

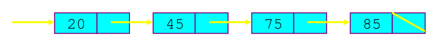


deleteEnd(85), deleteHead(20), deleteHead(45)

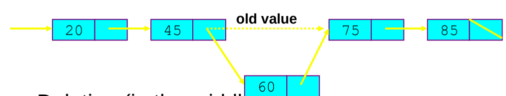


Linked Lists: Operations

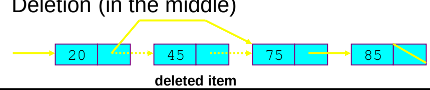
- Original linked list of integers:



- Insertion (in the middle):



- Deletion (in the middle)



Definition of linked list type:

```
struct Node{
     T data;
     Node* next;
     Node(const T & x, Node * p =NULL)
        :data(x),next(p){ }
};

We can also:

typedef Node* NodePtr;
```

Linked List Structure

- Creating a Node

  Node* p;

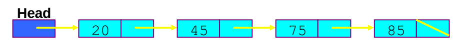
  p = new Node; //points to newly allocated memory

  p = new Node(12);
- Deleting a Node

  delete p;

– Access fields in a node
(*p).data; //access the data field
(*p).next; //access the pointer field
Or it can be accessed this way
p->data //access the data field
p->next //access the pointer field

Representing and accessing linked lists



- We define a pointer

```
Node* head;
```

that points to the first node of the linked list. When the linked list is empty then head is NULL.

Passing a Linked List to a Function

It is roughly the same as for an array!!!

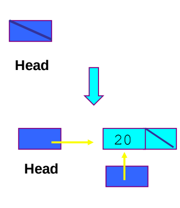
- When passing a linked list to a function it should suffice to pass the value of `head`. Using the value of `head` the function can access the entire list.
- Problem: If a function changes the beginning of a list by inserting or deleting a node, then `head` will no longer point to the beginning of the list.
- Solution: When passing `head` always pass it by reference or return the head by a function if the list `head` could be changed.

Implementation of an Linked List

Start the first node from scratch

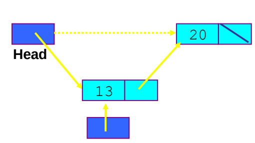
```
head = NULL;
```

```
Node* newPtr;

newPtr = new Node;
newPtr->data = 20;
newPtr->next = NULL;
head = newPtr;
```

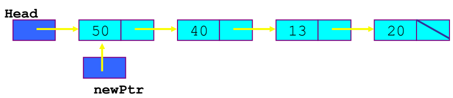


Inserting a Node at the Beginning

```
newPtr = new Node;
newPtr->data = 13;
newPtr->next = Head;
head = newPtr;
```



Keep going …



Adding an element to the head:

NodePtr&

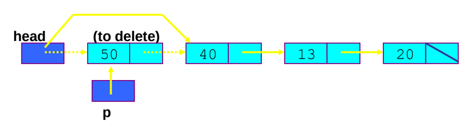
```
void addHead(Node*& head, int newdata){

  Node* newPtr = new Node;

  newPtr->data = newdata;
  newPtr->next = Head;
  head = newPtr;
}
```

Also written (more functionally, better!) as:

```
Node* addHead(Node* head, int newdata){

  Node* newPtr = new Node;

  newPtr->data = newdata;
  newPtr->next = Head;

  return newPtr;
}
```

Deleting the Head Node

```
Node* p;

p = head;
head = head->next;
delete p;
```



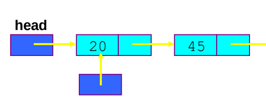

```
void deleteHead(Node*& head){

        if(head != NULL){
                NodePtr p = head;
                head = head->next;
                delete p;
        }
}
```

As a function:

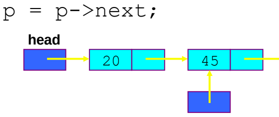
```
Node* deleteHead(Node* head){

        if(head != NULL){
                NodePtr p = head;
                head = head->next;
                delete p;
        }

        return head;
}
```

Displaying a Linked List

```
p = head;
```




```
p = p->next;
```



A linked list is displayed by walking through its nodes one by one, and displaying their data fields (similar to an array!).

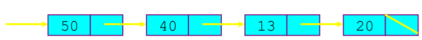
```
void displayList(Node* head){
        NodePtr p;
        p = head;
        while(p != NULL){
                cout << p->data << endl;
                p = p->next;
        }
}
```

Searching for a node

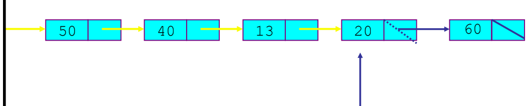
```
//return the pointer of the node that has data=item
//return NULL if item does not exist

Node* searchNode(Node* head, int item){
   NodePtr p = head;
   NodePtr result = NULL;
   bool found=false;
   while((p != NULL) && (!found)){
      if(p->data == item) {
              found = true;
              result = p;}
      p = p->next;
   }
   return result;
}
```

More operation: adding to the end

- Original linked list of integers:



- Add to the end (insert at the end):



Last element

The key is how to locate the last element or node of the list!

Add to the end:

```
void addEnd(NodePtr& head, int newdata){
   NodePtr newPtr = new Node;
   newPtr->data = newdata;
   newPtr->next = NULL;

   NodePtr last = head;
   if(last != NULL){      // general non-empty list case
      while(last->next != NULL)
              last=last->next;

      last->next = newPtr;
   }
   else      // deal with the case of empty list
      head = newPtr;
}
```

Link a new object to empty list

Link new object to last->next

Add to the end as a function:

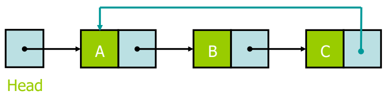
```
NodePtr addEnd(NodePtr head, int newdata){
   NodePtr newPtr = new Node;
   newPtr->data = newdata;
   newPtr->next = NULL;

   NodePtr last = head;
   if(last != NULL){        // general non-empty list case
      while(last->next != NULL)
              last=last->next;

      last->next = newPtr;
   }
   else// deal with the case of empty list
      head = newPtr;

   return head;
}
```

Adding a header node

- Adding an extra node as the beginning marker of a list makes coding easier.
- Insertion without a header node needs to distinguish insertion at the beginning and in the middle, in the former case, `head` needs to be changed:
- With a header node insertion is always done in the middle, and `head` never needs to be changed:

Variations of Linked Lists

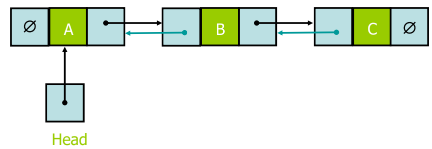
- *Circular linked lists*
  - The last node points to the first node of the list



  - How do we know when we have finished traversing the list? (Tip: check if the pointer of the current node is equal to the head.)

Variations of Linked Lists

- *Doubly linked lists*
  - Each node points to not only successor but the predecessor
  - There are two `NULL:` at the first and last nodes in the list
  - Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards



Array versus Linked Lists

- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.
  - **Dynamic**: a linked list can easily grow and shrink in size.
    - We don't need to know how many nodes will be in the list. They are created in memory as needed.
    - In contrast, the size of a C++ array is fixed at compilation time.
    - Vector can grow automatically, but expensive.
  - **Easy and fast insertions and deletions**
    - To insert or delete an element in an array, we need to copy to temporary variables to make room for new elements or close the gap caused by deleted elements.
    - With a linked list, no need to move other nodes. Only need to reset some pointers.

STL Linked List Implementation

- A template class list, a doubly linked list implementation;
- The associated iterator supports forward and backward traversal, which is a model of bidirectional iterator.
- Use list whenever linked list is needed, or when insertions and deletions are frequent operations.

Applications of Lists

- Polynomial operations: how polynomials can be represented and operations implemented. Try to design a class for polynomials.
- How can a polynomial be represented"

For example, $p = 5x^{20} + 3x^{6} - 5x^{2} + 20x + 1$

p is a list of terms, and a term consists of its coefficient and its exponent, and it is a pair of int, for example.

So, p is a list of pair of int, or p has the type of

vector<pair<int, int> >

- Is it a good structure for addition?

Strings（字符串）

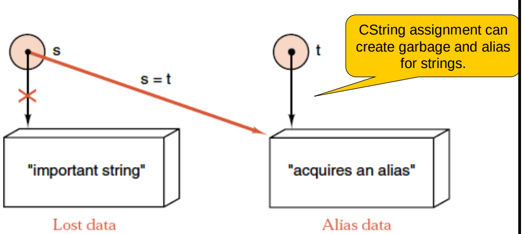
- A string is a sequence of characters. For example, "This is a string" and ""(empty string).
- A string ADT is the set of finite sequences of characters with the operations:
  - Length(str), returns the number of characters in str
  - Pos(str1, str2), the position of the first occurrence of str2 found in str1, or -1 if no match
  - Concat(str1, str2), a new string consisting of str1 followed by str2
  - Substr(str, l,m), a substring of length m starting at position l in str
  - Compare(str1,str2), Insert(str1,str2,i), and more.

Storage Structures for Strings

- A string ADT is a kind of list, but the operations are quite different from other lists.
- There are a number of ways to implementing strings:
  - As a fixed length array, the first element denotes the length of the string; used in Pascal;
  - As an array, but with the end of the string indicated using a special 'null' character '\0'; used in C

C-strings

- C-strings (strings in C) have type char *(字符数组);
- A string must terminate with '\0';
- C-strings is not implemented as an ADT.
- ✓ C-strings are widely available (<cstring> contains standard library functions);
- ✓ C-strings are efficient;
- × Not encapsulated;
- × Easy to misuse, may cause either garbage or aliases for string data;
- × Problem with uninitialized C-strings;




```
char *x = NULL; // uninitialized string
cout << strlen(x);
```

Problems! Can compile, but run-time error.

Safe Implementation of Strings

We can use encapsulation and embed the C-string representation as a member of the **class** String, including the features:

- Include the string length as a data member in the String class.
- The String class avoids the problems of aliases, garbage creation, and uninitialized objects by including an overloaded assignment operator, a copy constructor, a destructor, and a constructor.

```
class String {
public:                                   // methods of the string ADT
   String( );
   ~String( );
   String (const String &copy);           // copy constructor
   String (const char * copy);            // conversion from C-string
   String (List<char> &copy);             // conversion from List

   void operator = (const String &copy);
   const char *c_str( ) const;            // conversion to C-style string

protected:
   char *entries;
   int length;
};
```

Cstring

STL class string

- #include<string>
- Constructors
  - string();//construct an empty string,
    string s1;//s1 is empty
  - string(const char*s);
    //this string is initialized to contain a string copy of s
    string s2("data");
  - string(const string str, unsigned int pos=0, unsigned n=-1);
    string s3="data structures";
    string s4(s3), s5(s3,5,3);

string operator

- string& operator=(const string& str);
  - string s3=s2;
- char& operator[](unsigned int pos);

  /*Pre: pos<the number of char in this string.

  Post: a reference to the char that is at index pos in this string

  */
  - s2[0]='g';

string functions

- unsigned int length();

  //return the number of char in the string
- string substr(unsigned int pos=0, unsigned int n);

  // returns a substring from the pos and length n)

  string s1="data";

  string s2=s1.substr(1,2);

  string s3 = s1.substr(2);
- const char* c_str();

  /* returns a pointer to the first item of an array of size()+1 items, and whose first size() items are equal to the corresponding items of the string and whose last item is a null character.

  */

Summary

- Read Weiss Chapter 3;
- Understand ADTs and ADT list;
- Understand the differences between different implementations of the List ADT;
- Get used to use vector, list and string, and try to see how ADTs can be used to solve problems.
- Exercises: 3.2a, 3.4, 3.5, 3.11
- Set ADT: how sets can be represented and operations implemented. Try to design a class for sets, including membership, union and intersection operations.
- Exercise: can you generate an index table for a text?