Lecture 6 Feb 14, 2008

Goals for the day:

- Linked list – example 1 (similar to project)
- vector and list in STL (section 3.3)
- stack – implementation and applications

Overview of list (and comparison to array)

- list can be incrementally grown. (dynamic array resizing is expensive.)

- inserting next to a given node takes constant time. (In arrays, this takes O(n) time where n = size of the array.)

- searching for a given key even in a sorted list takes O(n) time. (in sorted array, it takes O(log n) time by binary search.)

- accessing the k-th node takes O(k) time. (in array, this takes O(1) time.)

Review of ADT

- Abstract Data Type (ADT)
- List ADT
- List ADT with Array Implementation
- Linked lists
- Basic operations of linked lists
  - Insert, find, delete, print, etc.
- Variations of linked lists
  - Circular linked lists
  - Doubly linked lists

Abstract Data Type (ADT)

- Data type
  - a set of objects + a set of operations
  - Example: integer
    - set of whole numbers
    - operations: +, -, x, /

- Can this be generalized?
  - (e.g. procedures generalize the notion of an operator)

Abstract data type
  - high-level abstractions (managing complexity through abstraction)
  - Encapsulation

Encapsulation

- Operation on the ADT can only be done by calling the appropriate function

- no mention of *how* the set of operations is implemented

➢ The definition of the type and all operations on that type can be localized to one section of the program

✓ If we wish to change the implementation of an ADT
  - we know where to look
  - by revising one small section we can be sure that there is no subtlety elsewhere that will cause errors

✓ We can treat the ADT as a primitive type: we have no concern with the underlying implementation

- ADT → C++: class
- oprations → C++: member function

Pros and Cons

- ✓ Implementation of the ADT is separate from its use
- ✓ Modular: one module for one ADT
  - Easier to debug
  - Easier for several people to work simultaneously
- ✓ Code for the ADT can be reused in different applications
- ✓ Information hiding
  - A logical unit to do a specific job
  - implementation details can be changed without affecting user programs
- ✓ Allow rapid prototying
  - Prototype with simple ADT implementations, then tune them later when necessary
- × Loss of efficiency

The List ADT

- A sequence of zero or more elements

$$A_1, A_2, A_3, \ldots A_N$$

- N: length of the list
- A_1: first element
- A_N: last element
- A_i: position i
- If N=0, then empty list
- Linearly ordered
  - A_i precedes A_{i+1}
  - A_i follows (succeeds) A_{i-1}

Typical Operations supported

- printList: print the list
- makeEmpty: create an empty list
- find: locate the position of an object in a list
  - list: 34,12, 52, 16, 12
  - find(52) $\rightarrow$ 3
- insert: insert an object to a list
  - insert(10,3) $\rightarrow$ 34, 12, 52, 10, 16, 12
- remove: delete an element from the list
  - remove(52) $\rightarrow$ 34, 12, 10, 16, 12
- findposn(k) : retrieve the element at position k
  - findposn(2) $\rightarrow$ 10

Some list functions

Consider the standard singly-linked list class:

```
class list {

private:
  Node* first;

list(int k) { first = new Node(k);}

void insert(int k, int posn) { . . . }

          . . .
}
```

We will add some functions for this class:

- Reverse the list
- Remove all the negative items from the list

Reverse the list

We want to reverse the list using the existing node of the list without creating new nodes.

Reverse the list

We want to reverse the list using the existing node of the list without creating new nodes.

```
void reverse() {
  if (head == NULL || head -> next == NULL) return;
   Node* p = head;
   Node* q = p-> next; p->next = NULL;
  while (q != NULL) {
    Node* temp = q -> next;
    q->next = p;
    p = q;
    q = temp;
   }
  head = p;
 }
```

Remove negative items from the list

Example:

List: -3, 4, 5, -2, 11 becomes 4, 5, 11

We will write this one recursively.

Remove negative items from the list

Example:

List: -3, 4, 5, -2, 11 becomes 4, 5, 11

We will write this one recursively.

```
void remove_negative() {
// removes all the negative items from a list
// Example input: -4 5 6 -2 8; output: 5 6 8
  if (head == NULL) return;
  else if (head->key >= 0) {
        List nList = List(head->next);
        nList.remove_negative();
    head->next = nList.head;
     }
  else {
         List nList = List(head->next);
    nList.remove_negative();
    head = nList.head;
    }
  }
```

A problem similar to Project # 1

Generate all the subsets of a given set of numbers.

Thus, if the input is `{1, 2, 4}` the output is:

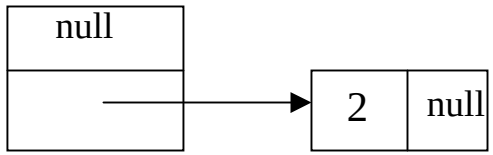
```
{}
{1}
{2}
{4}
{1, 2}
{1, 4}
{2, 4}
{1, 2, 4}
```

We will show how build works in this case.

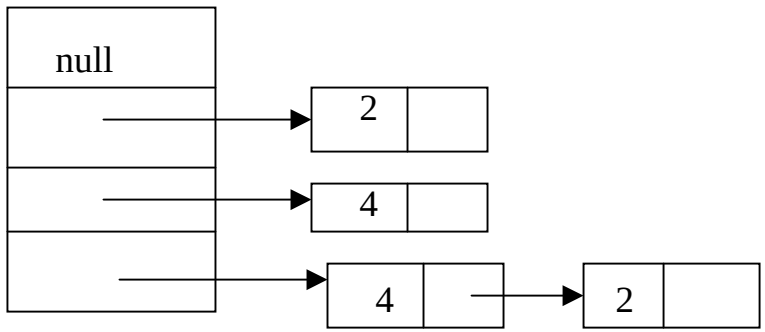
build(A, j) will generate all the subsets of the set {A[0], A[1], … , A[j – 1]}.

Thus, if A = { 2, 4, 6, 1}, then build(A, 1) will generate all the subsets of {2}, build(A, 2) will generate all the subsets of {2, 4} etc.

build(A, 1) ->



build(A, 2) ->



build (A, k) calls build(A, k – 1). Let temp be the result from the call.

It creates a copy of temp, say temp1.

It inserts A[k-1] into each list of temp1.

It merges temp and temp1 and returns the merged array of lists.

Concept of a static function

```
class List{
  private:
    Node* first;

  public:
   List ( ); // constructor
   ~List( ); // destructor
   void insert(int k); // insert k as the first item
   void print(); // print the list
   // some other functions
}
```

Suppose you need the following operation on the list:

Make all the terms of the list positive.

Thus, if L is: <-4, 5, 8, -11>, `L.make_positive()` would change L into <4, 5, 8, 11>

Code for make_positive will look like this:

```
void make_positive() {
  Node * temp = first;
  while (first != null)
      temp->key = change(temp->key);
 }
```

Code for change is:

```
  int change(int x) {
     if (x >= 0) return x; else
       return –x;
   }
```

We want to put change in the class List. But change is not addressed to the List object. In fact, it does not have anything to do with the List objects.

So you should call it static.

Static member

Suppose you want a data item that need not be created for each instance of the class, but just one data for the whole class.

For example, consider the node class:

```
class Node {
  private:
    int key; Node * next;
  public:
    Node(int k);
    ~Node();
    int get_key();
    void set_key(int k);
     . . .
 }
```

While running an application using the node class, we want to know how many nodes were created.

So, we need a variable count. But this variable does not belong to each instance of the class, but is shared by all the objects of the class.

Static member

```
static int count;
class Node {
  private:
    int key; Node * next;
  public:
    Node(int k){
     key = k; next = null;
     ++count;
    }
    ~Node();
    int get_key();
    void set_key(int k);
    static get_count() {return count;}
     . . .
 }
```

Static function – syntax for calling

Suppose C is a class, d is a data item (field) of C, f is a function that belongs to class C:

```
C x; // declare x as an object of type C
y = x.d // access the field d of x
z = x.f()  // calling the function f
```

Suppose C is a class, d is a static item of C, and f is a static function of C:

Inside C:

```
y = d // just refer to d without a qualifier object preceding it.
Z = f() //same for a function call.
```

Outside C:

```
y = C::d //Use the class name for scope resolution.

Z = C::f() //same for a function call.
```

Code for build

```
static set build(int a[], int s) {
  if (s == 0) {
   set x(1);  // set constructor shown in the next slude
   x.mems[0] = new List(); // list constructor also shown next
   x.size = 1; return x;
  }
 else {
   set L1 = build(a,s-1);
   set L2 = L1.copy();
   int s = L2.size;
   for (int k = 0; k < s; ++k)
    L2.mems[k]->insert(a[s-1]);
   set L = merge(L1, L2); // merge is also static
   return L;
  }
 }
};
```

Constructors for set and List

```
set(int n) {
  size = n;
  for (int j=0; j < n; ++j)
   mems[j] = null;
}


public:
List() {
  first = 0;
}
```

Main function for subsets construction

```
int main() {
 int s;
 cout << "Enter the size of the set." << endl;
 cin >> s;
 int a[s];
 cout << "Enter the elements of the set." << endl;
 for (int j=0; j < s; ++j)
   cin >> a[j];
 set temp = set::build(a, s);
 temp.print();
}
```