Topic 6

Dynamic array vs. linked list

資料結構與程式設計
Data Structure and Programming

10/30/2019

1

In the following topics,

we will introduce several special types of

Data Structures,

for example, list, array, set, map, hash, graph, etc.

Some people call them

Abstract Data Types (ADT)

or (an easier-to-understand name)

Container Classes

2

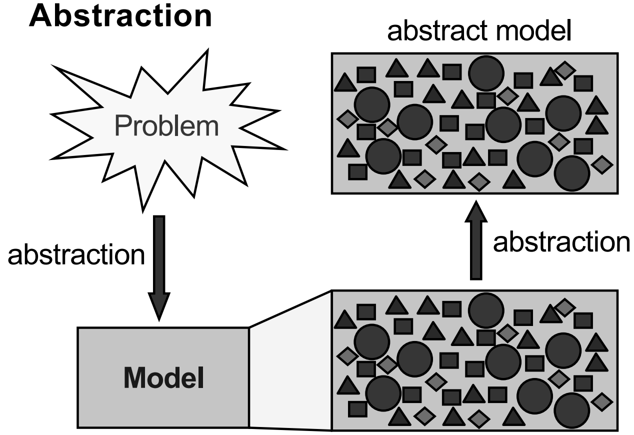
Pablo Picasso, “Accordionist”

Saver Containers

Abstract ?? Containers??

3

Abstraction



4

Data Types

- "A data type, as defined in many object-oriented languages, is a class"
  1. Data member
     - Define data
  2. Member functions
     - Define operations

So, what does the "Abstract" in "Abstract Data Type" mean?

Some Quotes about ADT...

- "...precisely specified independent of any particular implementation"
- "You don't know how the ADT computes, but you know **what** it computes"
- "The implementer of the class can change the implementation for maintenance, bug fixes or optimization reasons, without disturbing the client code"

ADT in Programming

- Obviously, these kinds of classes are not specific to any type of algorithms
  - In other words, they can be implemented independently of the algorithms that use them
- What they provide ---
  - Interface functions to operate on the data stored in the class
  - The implied complexity of these functions
- What they don't show (Abstracted away...) ---
  - What are the data members inside?
  - How the functions are implemented?

ADT in Programming

- That's why they are called "Abstract Data Types", or "Container Classes", and usually treated as special "utilities" for a programmer
  - Examples are:
    - List, array, queue, stack, set, map, heap, hash, string, bit vector, matrix, tree, graph, etc.
- The more and cleverer you use them, the better your program will be
  - That's the main purpose of learning this course

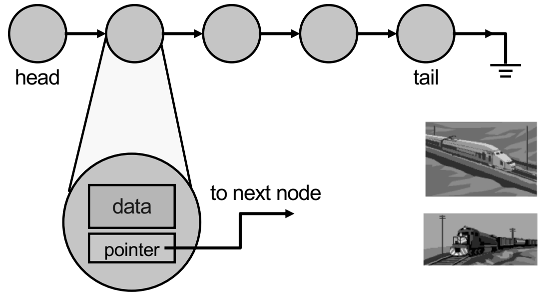
Classification of ADTs

1. Linear (Sequence) Data Types
   - List, array, queue, stack
2. Associative Data Types
   - Set, map, hash, heap
3. Topological Data Types
   - Tree, graph
4. Miscellaneous Types
   - String, bit vector, matrix

◆ Usually OOP programmer will implement these classes just once *(or adopt the existing ones)*, and later utilize them in various programs

Basic Concepts of Linked List

◆ An abstract data type in which the data are linked as a list



Linked List Implementation (I)

◆ Simple C-style implementation

```
struct MyStruct
{
   // define data here...
   int        _id;
   string     _name;

   // define the pointer here...
   MyStruct*  _next;
};

struct MyTop
{
   MyStruct*  _dataList;
   MyStruct*  _dataPointer;
};
```

data and pointer mixed together

list and pointer not distinguished

Linked List Implementation (II)

•Data encapsulation → Abstract Data Type
→ Like a container

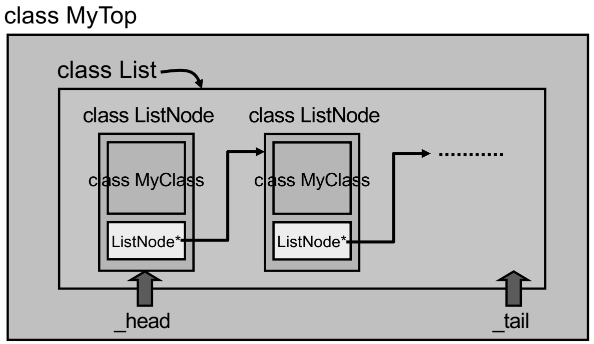
```
class MyClass
{
   // define data here..
   int        _id;
   string     _name;
};

class ListNode
{
   MyClass    _data;
   ListNode*  _next;
};

class List
{
   ListNode*  _head;
   ListNode*  _tail;
};

class MyTop
{
   List      _dataList;
   MyClass*  _dataPtr;
};
```

In other words...



More memory usage with data encapsulation??

◆ Simple C-style implementation

```
struct MyStruct
{
   int         _id;
   string      _name;

   MyStruct*   _next;
};
struct MyTop
{
   MyStruct*
            _dataList;
};
```

◆ C++ implementation with data encapsulation

```
class MyClass
{
   int        _id;
   string     _name;
};
class ListNode
{
   MyClass    _data;
   ListNode*  _next;
};
class List
{
   ListNode*  _head;
};
```

- However, whenever we need a list with different data type, we still need to define a new List class

Linked List Implementation (III)

◆ Template implementation

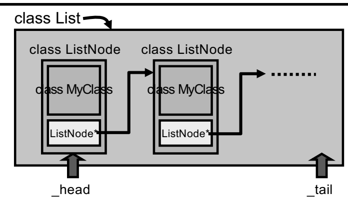
```
template <class T>
class ListNode
{
   T             _data;
   ListNode<T>*  _next;
};

template <class T>
class List
{
   ListNode<T>*  _head;
   ListNode<T>*  _tail;
};
```

One implementation multiple instantiations

```
List<int>      intList;
List<char>     charList;
List<MyClass>  myList;
...
```

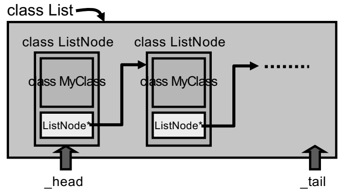

Complexity Analysis (Singly Linked List)




```
push_front(d) {
  ListNode* liNode
  = new ListNode(d);
  liNode->_next
  = _head;
  _head = liNode;
}
```

```
push_back(d) {
  ListNode* liNode
  = new ListNode(d);
  _tail->_next
  = liNode;
  _tail = liNode;
}
// Any corner case?
// What if "_tail"
is NOT known?
```

Complexity Analysis (Singly Linked List)




```
pop_front() {
  ListNode* liNode
  = _head;
  _head
  = _head->_next;
  delete liNode;
}
// What if "_head ==
NULL"?
```

```
pop_back() {
  .........
  delete _tail;
  .........
}
// How to find the
one in front of
_tail?
```


17

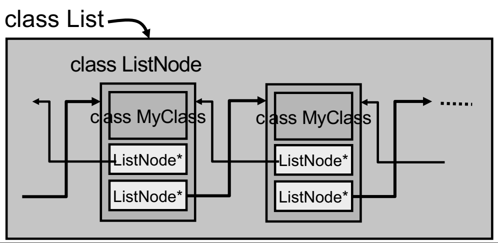
Complexity Analysis (Singly Linked List)

- push_front() O(1)
  push_back() O(1) // if tail is known, else O(n)
  pop_front() O(1)
  pop_back() O(n) // even if tail is known
  size() O(n) or O(1)
  empty() O(1) // complexity not equal to (size() == 0)
  insert(pos, data) O(n) (before pos) or O(1) (after pos)
  erase(pos) O(n)
  find(data) O(n)

18

Singly vs. Doubly Linked List

- Some operations, like "erase(node)", have linear complexity for singly linked list (Why?)
  - Don't know the previous nodes
- Doubly Linked List



19

Memory Overhead

- Assume (64-bit machine)
  - Pointer: 8 Bytes
  - Data: d Bytes
  - Total: n data
- Overhead = total memory – data memory
  - Data memory = d * n

1. Singly Linked List: (d + 8) * n + 8 * 2
   - Overhead = 8 * n + 16 (~ 8Bytes/data)
2. Doubly Linked List: (d + 16) * n + 8 * 2
   - Overhead = 16 * n + 16 (~ 16Bytes/data)

20

Complexity Analysis (Doubly Linked List)

push_front()	O(1)
push_back()	O(1)
pop_front()	O(1)
pop_back()	O(1)
size()	O(n) or O(1)
empty()	O(1) // != (size() == 0)
insert(pos, data)	O(1)
erase(pos)	O(1)
find(data)	O(n) ⟵

"Find" Operation

- One common way to speed up "find" operation is to keep the data always sorted
  - [Note] Binary Search: $O(\log_2 n)$

	10	100	1000	10K	100K
O(1)	1	1	1	1	1
$O(\log_2 n)$	4	7	10	14	17
O(n)	10	100	1000	10K	100K

- But, can we implement "binary search" using Linked List?

Why not?

Linear access

vs.

Random access

Access a ListNode & Traverse a List

```
template <class T>
class ListNode
{
   T              _data;
   ListNode<T>*   _next;
};
template <class T>
class List
{
   ListNode<T>*   _head;
   ListNode<T>*   _tail;
};
```

Does user need to know how List is implemented?

```
➔ for (ListNode<T>* node = myList.getHead();
       node != 0; node = node->getNext()) {
     ... }
```

why not "node != myList.getTail()"?

List Iterator

- In many standard List implementations, "class ListNode" is actually ***hidden*** from the user ---
  - Why should user know about the class "ListNode"?
    - User only interfaces with "class List"
    - The internal data field "ListNode*" is just one way of implementing "List"
- Use a generic interface class "List Iterator" to traverse a List

The Goal...

```
iterator li;
for (li = myList.begin();
     li != myList.end(); li++)...
--------- (compared to) ---------
ListNode<T> *n;
for (n = myList.head();
     n != 0; n = n->getNext();
```

➔ Overload "=", "!=", "++" for class iterator

List Iterator Implementation

```
class iterator {
// Conventionally, use lower case "i" for "iter..."
   ListNode<T>*   _node;

public:
   iterator(const ListNode<T>* const n = 0):
                         _node(n) {}

   const T& operator *() const;
   iterator& operator ++ ();
   iterator operator ++ (int);
   iterator& operator = (const iterator& i);
   bool operator != (const iterator& i) const;
};
```

➔ Act as a "wrapper class" for `ListNode<T>*`

But the question is:

"How to distinguish this generic iterator class from others?"

(iterators for Linked List, Array,… etc)

➔ One possible way is to declare it inside the "List" class

List Iterator Implementation (cont'd)

```
template <class T>
class List {
   ListNode<T>*    _head;
   ListNode<T>*    _tail;

   // Conventionally, use lowercase "i"
   class iterator {
      ListNode<T>*   _node;
   public:
      iterator(const ListNode<T>* const n = 0):
               _node(n) {}
   ...
   };

   // implicitly calling the iterator(_head) constructor
   iterator begin() { return _head; }
   iterator end() { return 0; }
};
```

Why return '0'? Is this a good implementation?

A List::iterator Example

```
int main() {
   List<int> intList;
   for (int i = 0; i < 10; ++i)
      intList.push_back(i * 2);

   List<int>::iterator li;
   for (li = intList.begin();
        li != intList.end(); li++) {
      cout << *li << endl;
   }
}
```


List<T>::push_back(const T& d)

```
void push_back(const T& d) {
   ListNode<T>* t
   = new ListNode<T>(d, 0);
   if (_tail != 0)
      _tail->setNext(t);
   else // _head = _tail = 0
      _head = t;
   _tail = t;
}
```

```
template <class T>
class ListNode {
  T          _data;
  ListNode<T>* _next;
};
```

[Question] Who frees the ListNode* memory?

Object or pointer data in a List

```
template <class T>
class ListNode {
  T            _data;
  ListNode<T>* _next;
};
```

- When the destructor of ListNode<T> is called, will the destructor of _data be called?
  - Yes, if _data is an object type
  - No, if _data is a pointer type
- Uh? Memory leak?
  - NO!! If _data is an object type, then it is a COPY of the data outside the List.
  - If it is a pointer, then it shares the same data storage. You can't delete it by List.

List<T>::pop_front()

```
void pop_front() {
   if (empty()) return;
   ListNode<T>* t = _head->getNext();
   delete _head;
   _head = t;
}
[Question] How about "_tail"?
           When should we care?

[Question] How about "_data" inside "_head"?
  Will it be destructed or "deleted"?
```


Destructors of List and ListNode

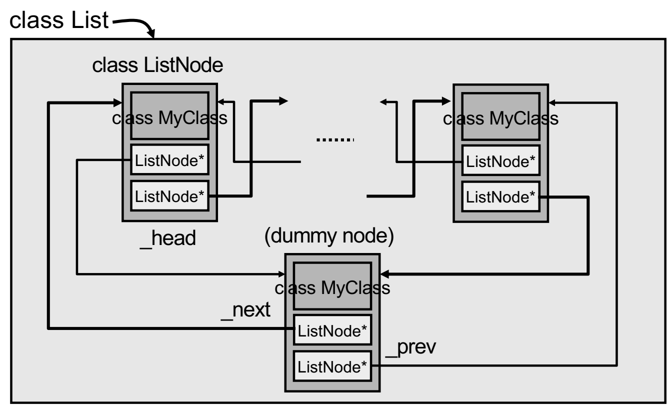
```
ListNode<T>::~ListNode() {
   // Do nothing.
   // But Will call the destructor of "T  _data"
   // But if "T" is a pointer type,
   // ➔ will not free the memory it points to.
}

List<T>::~List() {
   ListNode<T>* thisNode = _head;
   while (thisNode != 0) {
      ListNode<T>* nextNode = thisNode->getNext();
      delete thisNode;
      thisNode = nextNode;
   }
}
```


Note about the "end()"

- ◆ Remember, in STL, "end()" actually points to the next to the last node.
- ◆ In the previous example, we return '0' for "end()"
  - ➔ Any problem?
  - Potential misjudgment on "n == end()"
  - How to do backward traversal?
- ◆ The solution in HW#5 (also in STL's list<T>)
  - Create a dummy ListNode<T>* as the end

Dummy ListNode<T>* as the end()



Dummy ListNode<T>* as the end()

◆ Things to consider...

1. What happens when the List<T> is just constructed?
2. size(), empty()?
3. push_back(), push_front()
   ➔ need to properly update _head, _tail
4. pop_back(), pop_front()
   ➔ what happen if it has just one element or is empty?
5. Do we need "_tail"?

Sorting in Linked List

◆ As we say, since the iterators in linked list are not randomly accessible, it's not possible to implement binary search on it.

◆ Sorting on Linked List: $O(n^2)$
- Bubble sort, selection sort, etc.

Classification of ADTs

1. Linear (Sequence) Data Types
   - List, array, queue, stack
2. Associative Data Types
   - Set, map, hash, heap
3. Topological Data Types
   - Tree, graph
4. Miscellaneous Types
   - String, bit vector, matrix

◆ Usually OOP programmer will implement these classes just once *(or adopt the existing ones)*, and later utilize them in various programs

Array vs. List

◆ In many programmers' view, "array" is less favorable than "list" because they think the array class is ---

1. Limited in size (i.e. array bound)
2. Expensive in "erase" operation
3. No clear advantage other than "random access by index"

➔ That's because they don't know enough about "Dynamic Array"

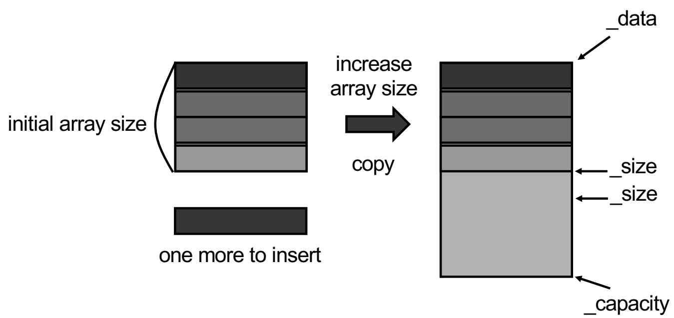
Static Array

- Array with fixed size   // e.g.   int arr[100];
- "Insert/erase()" operation
  - O(1) if inserted at the end
  - If the element order is not important
    - O(1) insert anywhere (how?)
    - O(1) erase
  - If the element order does matter
    - O(n) insert at the beginning or anywhere
    - O(n) erase
    - ➔ Is this common? (comparing to list...)
- "Find()" operation
  - Can have $O(\log_2 n)$ complexity (how?)

Static vs. Dynamic Array

- Static array is indeed limited in usage, and may create memory problems
  - Not recommended in general
- Dynamic array removes the array size limitation, and when compared to linked list, its performance (runtime and memory) is actually much better
  - Highly recommended

Basic Concept of Dynamic Array



Dynamic Array Implementation

```
template <class T>
class Array
{
   T*        _data;
   size_t    _size;
   size_t    _capacity

public:
   Array(size_t t = 0)
   : _size(t), _capacity(t) {
      _data = initCapacity(t);
   }
};
```

"Size" in Dynamic Array

- `[Note] In previous example, _size = t, not 0`
  - ➔ `follow the semantics of STL`
  - `We can access array[0 ~ (t-1)] after construction`
- [compare]
  - Array<int> arr1; // size = 0
    arr1[0] = i; // Error!!
    arr1.push_back(i); // OK; size becomes 1
  - Array<int> arr2(10); // size = 10
    arr2[0] = i; // OK
    arr2.push_back(j); // What's the size now?

"Capacity" in Dynamic Array

- Initialized in array constructor
- When _size == _capacity, how to grow?
  - ➔ Doubled (e.g. 2➔4, 3➔6, 5➔10, etc)
  - Issue: How to do memory management?
    - Remember: difficult to recycle if different in size

  [Sol#1] Powered of 2 in memory allocation
  - Issue: waste memory
    - Many arrays may have size < 10, but only have capacity choices as {2, 4, 8, 16 }

  [Sol#2] Hybrid (1, 2, 3, ...7, 8, 16, ..., 2^n, ...)

Important Member Functions for Array

```
1. T& operator [] (size_type i);
2. const T&
   operator [] (size_type i) const;
3. void push_back(const T& d) {
      if (_size == _capacity)
         expand();
      data[_size++] = d;
   }
4. void resize(size_type s);
```

// s can be smaller or larger than _size

Complexity Analysis (Dynamic Array)

- push_front() O(n) or O(1) // if order not matters
  push_back() O(1)
  pop_front() O(n) or O(1) // if order not matters
  pop_back() O(1)
  size() O(1) // not O(n), why?
  empty() O(1)
  insert(pos, data) O(n) or O(1) // if order not matters
  erase(pos) O(n) or O(1) // if order not matters
  find(data) O(n) or O(log n) // why?

If order does not matter, almost all operations are O(1)!!

Memory Overhead of Dynamic Array

- Assume (64-bit machine)
  - Pointer: 8 Bytes
  - Data: d Bytes
  - Total: n data
- Overhead = total memory – data memory
  - Data memory = d * n
- Dynamic Array Overhead = 24 Bytes only (why??)
  - (cf) Singly Linked List = 8 * n + 16
  - (cf) Doubly Linked List = 16 * n + 16

The Data in the Array Can be Sorted

- Option #1 (dynamic)
  - Whenever a data is inserted, update the array so that the elements are in right order
  - O(log n) in finding the place to insert; O(n) in updating the array

    ➔ Inserting n elements ➔ $O(n^2)$ // NOT O(n log n)

    ➔ Array may not be the best ADT

    ➔ In such case, "balanced binary search tree (BST)" (e.g. STL Set/Map) should be better
- Option #2 (static)
  - If we care about the order only after all the elements are inserted

    ➔ Sorted only once

    ➔ Inserting n elements ➔ O(n log n)
  - Has the same "find()" complexity as "set" or "map", but much less runtime and memory overhead than BST!!

Some notes about the Array<T> in HW#5

- Don't worry about sorting for Array<T>, we call STL:
  - void sort(RandomAccessIterator first,
    RandomAccessIterator last,
    StrictWeakOrdering comp);

    ➔ as lone as operator < is overloaded, you can use sort()
- No need to implement class ArrayNode<T>. Why??
- The capacity always grows from: 0 → 1 → 2 → 4 → 8 → ... → 2^n

Performance Comparison: Dynamic Array vs. Linked List

- Task 1
  1. Insert n data (1 by 1)
- Task 2
  1. Insert n data (1 by 1)
  2. Destroy the ADT (remove all)
- Task 3
  1. Alternatively insertions and deletions
- Task 4
  1. Sort the data

(Try different scenarios and report in HW #5)

“vector” and “list” in STL

- In fact, many wrapper classes around the real data members
- In essence…
  - class vector {
    T* _M_start;
    T* _M_finish;
    T* _M_end_of_storage;
    };
  - class list {
    std::_List_node_base *_M_node;
    };
    class _List_node_base {
    std::_List_node_base *_M_next;
    std::_List_node_base *_M_prev;
    };

Other Linear ADT

1. Queue (also known as FIFO)
2. Stack (also known as FILO)

- Use “adaptor class” to implement on top of other linear ADT
  - For example,

```
template <class T, class C = Array<T> >
class Stack {
   C    _elements;
public:
   // only define operations
   // that make sense to "stack"
   // e.g. push(), pop(), top(), etc
};
```