cpt102 整理

Intro

理解数据结构的两种视角：

- 数学和逻辑视角（抽象）
- 代码实现视角（具体）

第一个视角为ADT(抽象数据类型)，后一个视角为implements

整理方式：

1. 逻辑和抽象层面
2. 操作
3. cost
4. implements and code

两种结构：链式储存vs连续储存

简略地说，前者易添加（创建新节点并引用），弱查找（遍历前续节点）；后者易查找（寻址），弱添加（扩容）。取决于场景中需要的操作量。

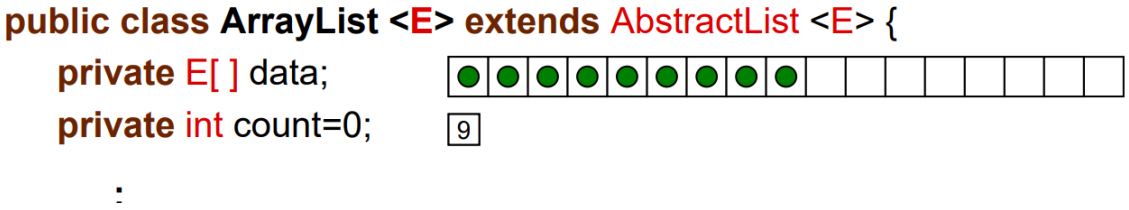
链式: Linked List, Linked stack, Linked Queue

数组：Array List, Array set

ArrayList（lec9）

ArrayList 类是一个可以自动扩容的数组，与普通数组的区别就是它是没有固定大小的限制。

数组：分配了连续内存/地址的一块空间，可以通过[index]指定需要的地址。



重要操作：get, set, remove, add

- `get(int index)`， 得到index对应的item并返回，复杂度O（1）

```
public E get(int index){
    if (index < 0 || index >= count){
        throw new                IndexOutOfBoundsException();
    }
    return data[index];
}
```

- `set(int index, E value)`， 更改index的item为value，返回。O(1)

```
public E set(int index, E value){
    if (index < 0 || index >= count)  throw new
IndexOutOfBoundsException();
    E ans = data[index];
    data[index] = value;
    return ans;
}
```

- `remove(int index)` 移除index的item。然后把后面的向前依次移位，补充空位。O(n)

```
/** Removes the element at the specified index, and returns it.
 * Throws an IndexOutOfBoundsException if index is out of
 bounds */ public E remove (int index){
   if (index < 0 || index >= count)
     throw new IndexOutOfBoundsException();
   E ans = data[index];                  ←remember
   for (int i=index+1; i< count; i++)←move items down
     data[i-1]=data[i];
   count--;                              ←decrement
   data[count] = null;                   ←delete previous last element
   return ans;                           ←return
 }
```

- `add(int index, E value)` 加入alist并依次后移。在移位之前加入 `ensureCapacity()` 用来检测大小够不够用，不够的话自动扩容。O(n)

```
/** Adds the specified element at the specified index.*/
public void add(int index, E item){
    if (index < 0 || index > count)      ←can add at end?
      throw new IndexOutOfBoundsException();
    ensureCapacity();                    ←make room
    for (int i=count; i > index; i--)    ←move items up
      data[i]=data[i-1];
    data[index]=item;                    ←insert
    count++;                             ←increment
  }
```

notes：这里用数组的朴素扩容，即创建一个新数组进行数组复制，然后改变引用指向这个新数组。一个时间优化：超出数组大小后，进行倍数扩容（Geometric Resizing），用乘数代替加数扩容。

```
private void ensureCapacity () {
    if (count < data.length) return;      ← room already
    E [ ] newArray = (E[ ]) (new Object[data.length * 2]);
    for (int i = 0; i < count; i++)       ← copy to new array
        newArray[i] = data[i];
    data = newArray;                      ← replace
}
```

- get O(1)
- set O(1)
- remove O(n)
- add (at i) O(n) (worst and average)
  (have to shift up
  may have to double capacity)
- add (at end) O(1) (most of the time)

  (when doubles cap) O(n) (worst)

  O(1) (amortised average)
  (if doubled each time)

Analysis cost（lec10)

对于程序的时间和空间复杂度分析.

Benchmarking: program cost

计算机械时间，实际的电脑运行时间和内存量

`System.currentTimeMillis()` ‘

→ time from system clock in milliseconds

Analysis: Algorithm complexity

操作的次数，占用的内存数量，考虑“steps”

渐进分析

算法知识，采用big-o表示。

一些选择step的方法：

- 只考虑最坏（数据最多）的情况
- 选择具有代表性的操作，比较和数组访问比数字相加更有代表性

```
public  E  remove (int index){
    if (index < 0 || index >= count) throw new ....Exception();
    E ans = data[index];
    for (int i=index+1; i< count; i++)      ← in the innermost loop
      (data[i-1]=data[i];)                  ←Key Step
     count--;                               Each for loop:
    data[count] = null;                     1 comparison: i<count
    return ans;                             1 addition: i++
  }                                         1 data retrieval: data[i]
                                            1 subtraction: i-1
                                            1 memory store: data[i-1]=data[i]
```

Array Set

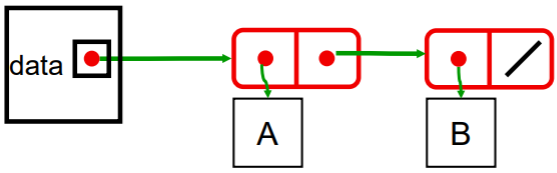
同arraylist，不要求数组的顺序，不可以有重复值。

操作：

- contains(item)
- add(item) always at end
- remove(item) ← only move last item down

链表（lec12-14）

array list的问题：添加item较慢，空间占用可能冗余。引入Linked List。



链表的节点（node）由一个存储数据元素的**数据**， 和存储下一个节点地址的**指针**组成。

```
public class LinkedNode <E>{
    private E value;
    private LinkedNode<E> next;
    public LinkedNode(E item, LinkedNode<E> nextNode){
        value = item;
        next = nextNode;
    }
```

链表的操作：

get、set

从头节点开始遍历链表，直到找到这个node。O(n)

```
public E get(int index){
    if (index < 0) throw new IndexOutOfBoundsException();
    Node<E> node=data;
    int i = 0;  // position of node
    while (node!=null && i++ < index)  node=node.next;
    if (node==null)   throw new IndexOutOfBoundsException();
    return node.value;
}
```

```
public E set(int index, E value){
    if (index < 0) throw new IndexOutOfBoundsException();
    Node<E> node=data;
    int i = 0;  // position of node
    while (node!=null && i++ < index) node=node.next;
    if (node==null)   throw new IndexOutOfBoundsException();
    E ans = node.value;
    node.value = value;
    return ans;
}
```

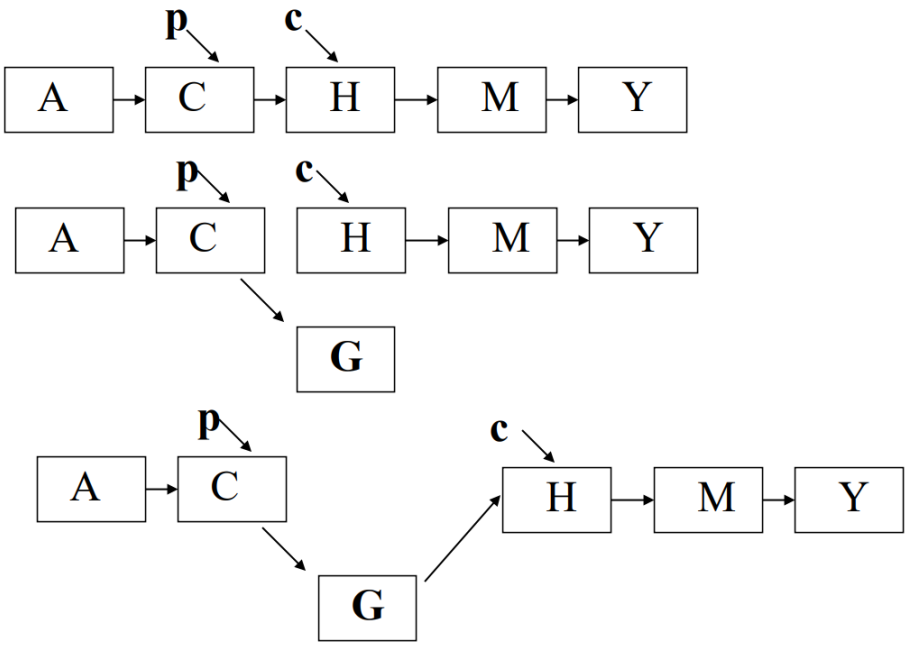
Same as get

set多了一个赋值操作，用于更改value

add

在index位置添加一个新节点。如果插入在开头，则直接创建一个指向O(1).

非开头 O(n)




```
public void add(int index, E item){
    if (item == null) throw new IllegalArgumentException();
    if (index==0){              // add at the ht.
        data = new Node(item, data);
        count++;
        return;
    }
    Node<E> node=data;
    int i = 1;                  // position of next  node
    while (node!=null && i++ < index) node=node.next;
    if (node == null) throw new IndexOutOfBoundsException();
    node.next = new Node(item, node.next);
    count++;
    return;
}
```

remove

删除值等于item的节点. O(n)

```
public boolean remove (Object item){
    if (item==null || data==null) return false;
    if (item.equals(data.value))   // remove the front item.
        data = data.next;
    else {          // find the node just before a node containing the item
        Node<E> node = data;
        while (node.next!=null && !node.next.value.equals(item))
            node=node.next;
        if (node.next==null)   return false;   // off the end
        node.next = node.next.next;   // splice the node out of the list
    }
    count--;
    return true;
}
```