

数据结构与算法

DATA STRUCTURE

第五讲 动态数组和链表

胡浩栋

信息管理与工程学院

2018 - 2019 第一学期

课堂内容

- 回顾什么是数据结构
- 线性数据结构
 - （静态和动态）数组
 - （动态）链表

数据结构的目的:

- 维护一个动态的数据集合，使得
 - 有效地存储数据元素
 - 支持快速的插入/删除/查找

• INSERT

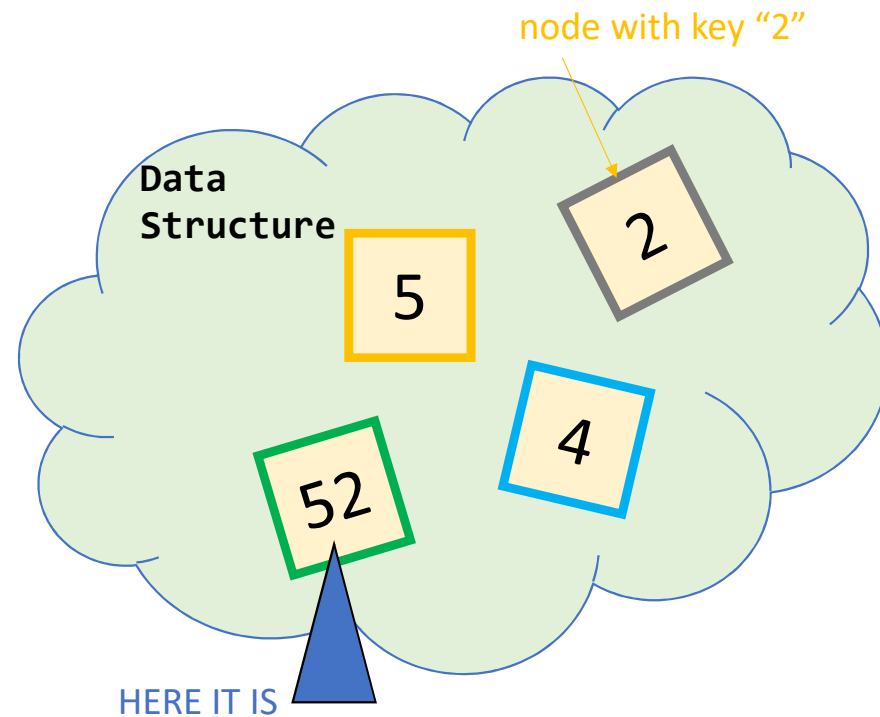
5

• DELETE

4

• SEARCH

52



数据结构核心问题

1. 查找`search`，时间复杂度
2. 插入`insert`，时间复杂度
3. 删除`delete`，时间复杂度
4. 数据元素的存储方式，空间复杂度

基本线性数据结构

- 数组Array（静态和动态）
- 链表Linked List

静态结构

- 不能直接插入/删除，比如
 - C++内置数组
 - `std::array`是静态数组结构
- 一般来说，链表Linked List是动态的

[C++11] for-each语句

- for-each语句

```
for (element_declaration : array)
    statement;
```

- 这里fn不是下标，fn是对fibNums[]数组元素的复制

```
int fibNums[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };
for (int fn : fibNums)
{
    cout << fn << " ";
}
```

- 可以直接和auto搭配使用，更方便

```
for (auto fn : fibNums)
{
    cout << fn << " ";
}
```

for-each和引用

- 如果加上引用 (&)，不再传值，可以修改原数组值

```
int array[5] = { 9, 7, 5, 3, 1 };  
for (auto &element: array)  
{  
    std::cout << element << ' ';  
    element = 0;  
}
```

- 如果即要避免元素值复制，又不要修改原数据，用const reference

```
for (const auto &element: array)  
{  
    std::cout << element << ' ';  
    element = 0; // ERROR: element is readonly  
}
```


for-each语句

- 不能用在数组形参上，因为这个形参就是指针，不知道数组长度
- 所以参数形式最好改成指针，避免误解

```
int sumArray(int array[])
{
    int sum = 0;
    for (const auto &number : array) // ERROR!!!
    {
        sum += number;
    }
    return sum;
}
```

std array in C++11

- 固定长度（静态）数组 `#include <array>`

```
#include <array>

std::array<int, 3> myarray;
```

- 初始化

```
array<int, 5> myarray = { 9, 7, 5, 3, 1 };
array<int, 5> myarray2 { 9, 7, 5, 3, 1 };
```

```
array<int, > myarray = { 9, 7, 5, 3, 1 }; // illegal, array length must be provided
```

- 赋值

```
std::array<int, 5> myarray;
myarray = { 0, 1, 2, 3, 4 }; // okay
myarray = { 9, 8, 7 }; // okay, elements 3 and 4 are set to zero!
myarray = { 0, 1, 2, 3, 4, 5 }; // not allowed, too many elements in initializer list!
```

std array in C++11

- 用下标运算符[]取值，不检查越界

```
myarray[2] = 6;
```

- 用at()取值，检查越界

```
std::array<int, 5> myarray { 9, 7, 5, 3, 1 };  
myarray.at(1) = 6; // array element 1 valid, sets array element 1 to value 6  
myarray.at(9) = 10; // array element 9 is invalid, will throw error
```

- Size()取长度
- Sort()

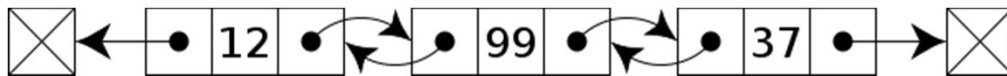
```
std::sort(myarray.begin(), myarray.end());
```

- 遍历

```
for (const auto &element : myarray)  
    std::cout << element << ' ';
```

动态结构：插入/删除/查找

- `std::vector` 动态数组，按动态数组来实现
数据元素都在连续的内存块，
好处是可以直接寻址，空间小，增加了data locality
增加数据元素可能需要重新分配，插入删除费时
- `std::list` 动态链表，按双链表来实现
数据元素离散分布，用指针链接，查找费时
插入/删除复杂度很低 $O(1)$



std vector in C++03

- 动态数组 `#include <vector>`
- 初始化

```
std::vector<int> array;  
std::vector<int> array2 = { 9, 7, 5, 3, 1 };  
std::vector<int> array3 { 9, 7, 5, 3, 1 }; /
```

- C++11赋值。数组是动态的

```
array = { 0, 1, 2, 3, 4 }; // okay, array length is now 5  
array = { 9, 8, 7 }; // okay, array length is now 3
```

std vector in C++03

- 用下标运算符[]取值，不检查越界

```
array[6] = 2; // no bounds checking
```

- 用at()取值，检查越界

```
array.at(7) = 3; // does bounds checking
```

- Size()取长度，Resize()改变长度

- 遍历

```
std::vector<int> array { 0, 1, 2 };  
array.resize(5); // set size to 5  
  
std::cout << "The length is: " << array.size() << '\n';  
  
for (auto const &element: array)  
    std::cout << element << ' ';
```

- 自动清理内存，不用担心泄露

std list in C++03

- 动态双链表 `#include <list>`
- 初始化

```
// Create a list containing integers
std::list<int> l = { 7, 5, 16, 8 };
```

- `push_back`, `pop_back` 动态操作

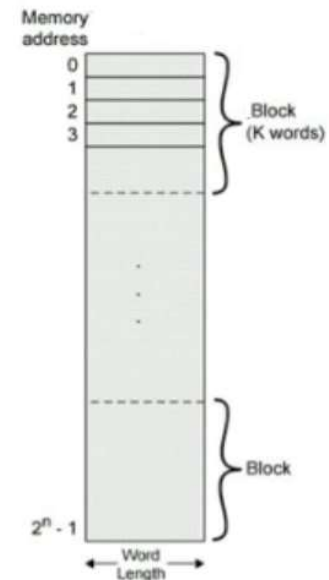
```
// Add an integer to the front of the list
l.push_front(25);
// Add an integer to the back of the list
l.push_back(13);
```

- `Size()` 取长度, `Resize()` 改变长度
- `Sort()`
- 遍历

```
// Insert an integer before 16 by searching
auto it = std::find(l.begin(), l.end(), 16);
if (it != l.end()) {
    l.insert(it, 42);
}
```

数据的局部特性Data locality

- 属于计算机里一种可以预测的行为
 - 一次IO读取的利用率, **block size**
 - 缓存**caching**, 包括时间**locality**, 空间**locality**
 - 预取操作**prefetching**
 - CPU分支预测
- 大数据上一般用数据块读取次数来衡量时间复杂度, 而不是单个数据节点的操作次数。
- 研究数据的存放方式来增加**data locality**是一个研究方向



接下来,

- 动态数组
 - `std::vector`的底层结构
 - 不讲`sequential list`, 和动态数组差不多, 后面数据结构不适用
- 单、双、循环链表
 - `std::list`的底层结构
 - 简单讲解单链表、循环链表

动态数组Dynamic Arrays

- $O(n)$ 插入/删除:

6



- SortedArray $O(\log(n))$ 查找, $O(1)$ 选择:



查找: Binary search to see if 3 exists

二分法查找 Binary search 在下一堂课里还会详细讲解

选择: Third element is 3

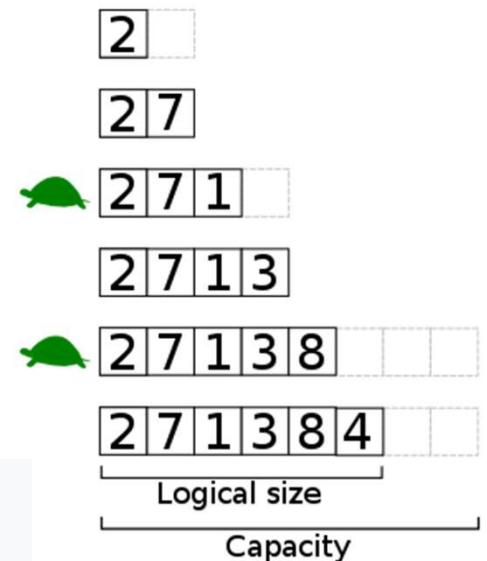
动态数组Dynamic array

- 前面IntArray其实是静态结构，数组大小是固定的
- 如何动态支持插入/删除
- 最naive的做法：插入一个元素后重新分配一次内存

插入算法

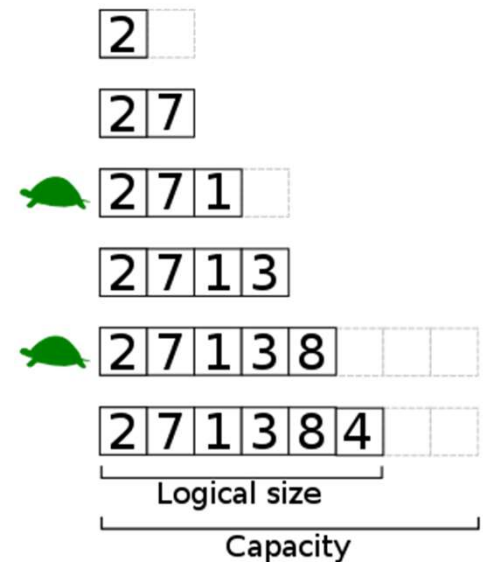
- 引入另外一个私有变量capacity，区别于size
- 算法是在数组size达到capacity时候，重新分配2倍大小的新数组
- 把旧数组元素复制到新数组
- 正常情况是把要插入的位置后面的数据元素往后移动一个位置，然后再插入

```
function insertEnd(dynarray a, element e)
    if (a.size == a.capacity)
        // resize a to twice its current capacity:
        a.capacity ← a.capacity * 2
        // (copy the contents to the new memory location here)
    a[a.size] ← e
    a.size ← a.size + 1
```



删除算法

- 算法是在数组size达到capacity/4时候，重新分配一半大小的新数组
- 把旧数组元素复制到新数组
- 正常情况是把要删除元素后面的数据元素往前移动一个位置，保持连续性。

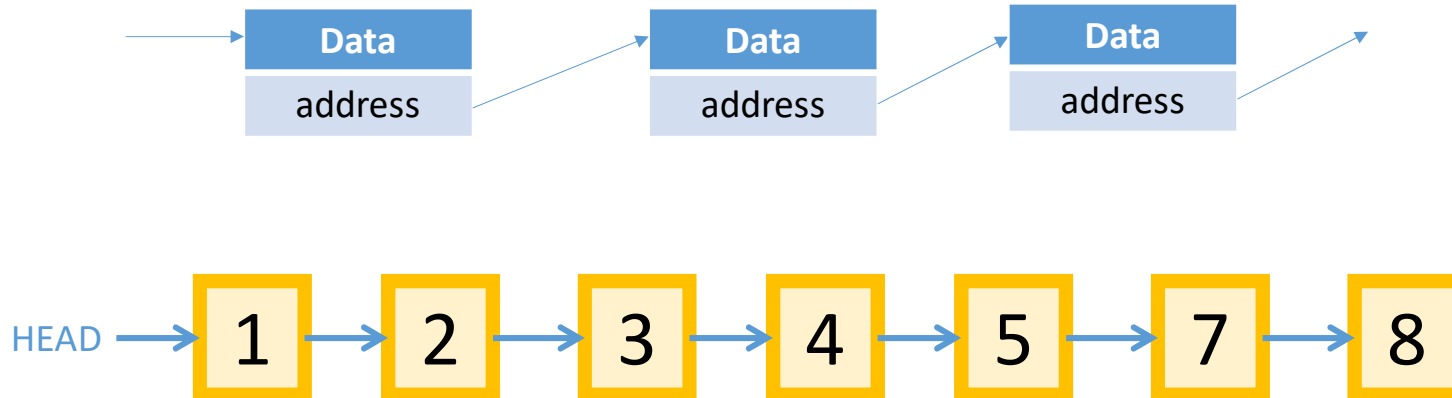


作业

- 在IntArray基础上实现插入Insert/删除Delete算法
 - `void Insert(int index, int value);`
 - `void Delete(int index);`
- 注意原来的_size变成逻辑数组大小，真正的数组内存大小应该是_capacity

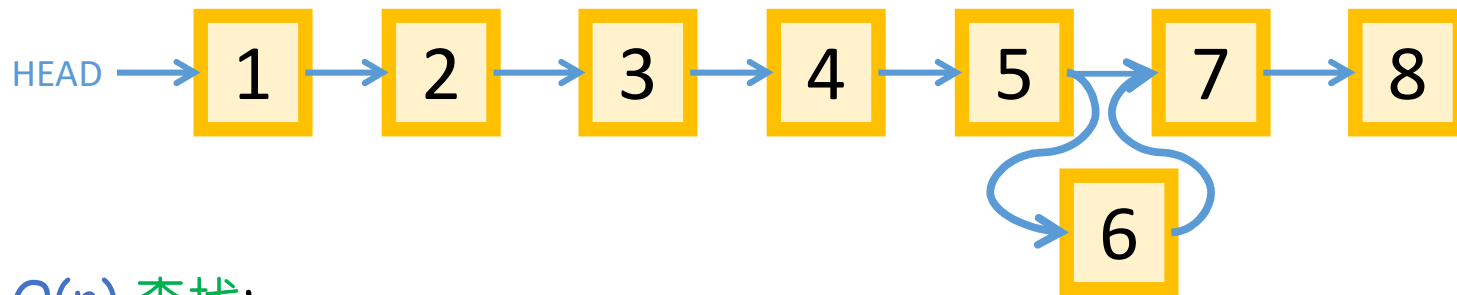
Linked list

- 一串链接起来的数据节点
- 在内存里是离散分布
- 每个节点除了数据，还有下一个节点的地址

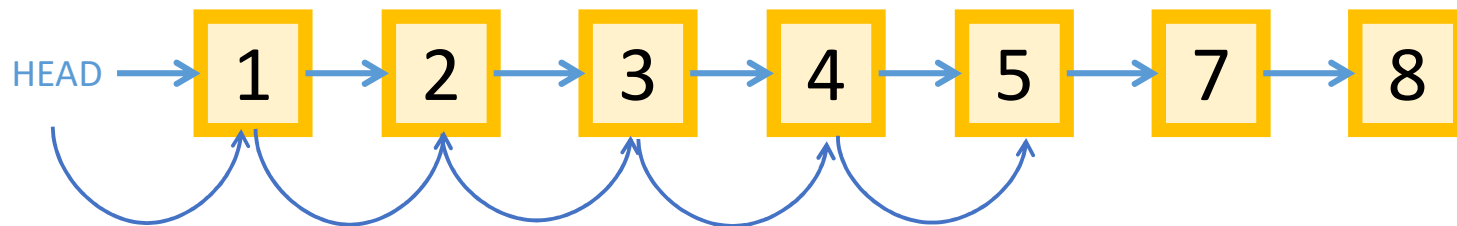


Linked list

- $O(1)$ 插入/删除 (假如我们有了插入位置地址的指针):

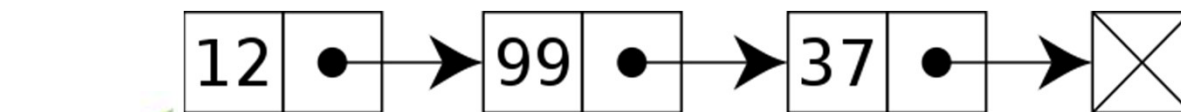


- $O(n)$ 查找:



单链表Linked list

- 一组链接起来的结构体数据，
 - 每个节点除了数据，还有下一个节点的地址
 - 在内存里是离散分布（数组是连续的空间）
 - 有一个头节点，每个节点只指出后继节点（next指针）



Head



```
struct ListNode
{
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(nullptr) {}
};
```

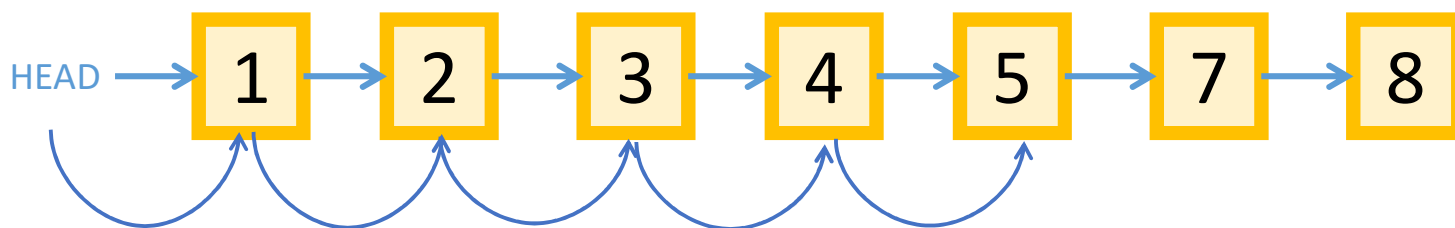
查找

- 从头节点开始，看下一个是不是要查找的节点。
- 时间复杂度 $O(n)$

```
ListNode* Search(ListNode *head, int val)
{
    if (head == nullptr)
    {
        return nullptr;
    }

    ListNode *pNode = head;
    while(pNode)
    {
        if (pNode->val == val)
        {
            return pNode;
        }
        pNode = pNode->next;
    }

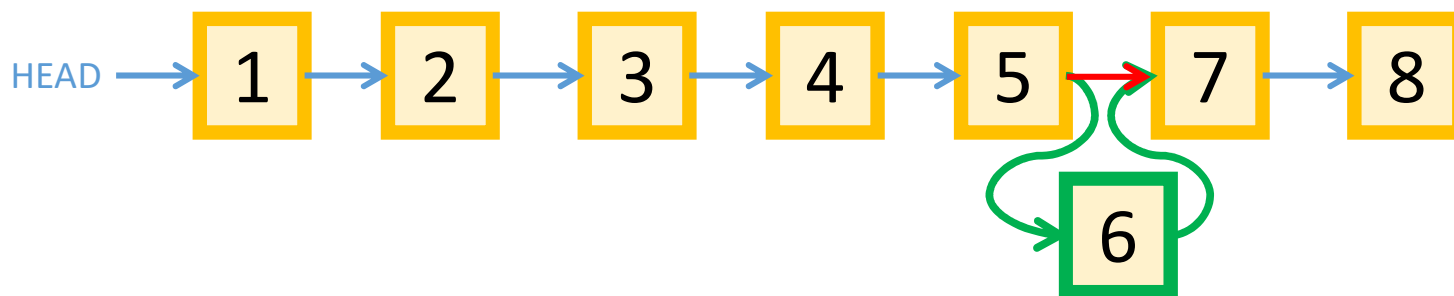
    return nullptr;
}
```



插入

- 要插入一个节点，做法就是把当前节点指向新节点，然后新节点指向下一个节点
- 时间复杂度 $O(1)$ ，前提是你知道要插入节点的地址

```
ListNode* Insert(ListNode *node, int val)
{
    ListNode *newNode = new ListNode{val, nullptr};
    if (node != nullptr)
    {
        newNode->next = node->next;
        node->next = newNode;
    }
    return newNode;
}
```

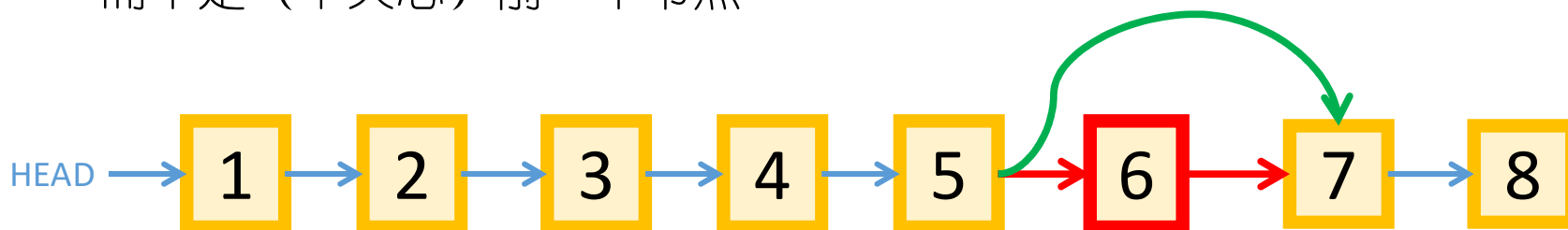


删除

- 要删除一个节点，一般算法是
 - 先找到前一个节点，
 - 然后让前一个节点指向后一个节点
- **缺点**是用户调用Delete时候只提供要删除的元素（节点），而不是（不关心）前一个节点

```
void Delete(ListNode *prev)
{
    if (prev == nullptr || prev->next == nullptr)
    {
        return;
    }

    ListNode *curr = prev->next;
    prev->next = curr->next;
    delete curr;
}
```

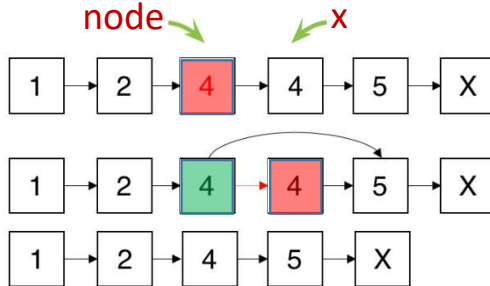


删除算法*

- 如果只有要删除的元素（或节点）



- 改进的做法是如下图，不用查找前一个节点，除非删除的是最后一个节点



- 再改进：在链表最后额外添加一个尾节点，比如设置值为INT_MIN，这样删除的节点一定不是最后一个节点

```
void Delete(ListNode *node)
{
    if (node == nullptr || node->next == nullptr)
    {
        return;
    }

    ListNode *x = node->next;
    node->val = x->val;
    node->next = x->next;

    delete x;
}
```

建立单链表

- 定义头指针：

```
struct ListNode
{
    int val;
    ListNode *next;
};

ListNode *head = new ListNode{0, nullptr};
```
- 定义尾指针

```
ListNode *tail = head;
```
- 调用插入算法，把新节点插入在尾指针之后

```
char ch;
while ((ch = cin.get()) != 'Q')
{
    Insert(tail, (int)ch);
    tail = tail->next;
}
```

单链表输出

- 从头节点开始，依次遍历后继节点

```
void Print(ListNode *nd)
{
    while (nd)
    {
        if (nd->val < INT_MAX)
        {
            cout << nd->val << '\t';
        }
        nd = nd->next;
    }
}
```

完整实现

```
struct ListNode
{
    int val;
    ListNode *next;
};

ListNode* Search(ListNode *head, int val)
{
    if (head == nullptr)
    {
        return nullptr;
    }

    ListNode *pNode = head;
    while (pNode)
    {
        if (pNode->val == val)
        {
            return pNode;
        }
        pNode = pNode->next;
    }

    return nullptr;
}
```

```
void Delete(ListNode *node)
{
    if (node == nullptr || node->next == nullptr)
    {
        return;
    }

    ListNode *x = node->next;
    node->val = x->val;
    node->next = x->next;

    delete x;
}

ListNode* Insert(ListNode *node, int val)
{
    ListNode *newNode = new ListNode(val, nullptr);
    if (node != nullptr)
    {
        newNode->next = node->next;
        node->next = newNode;
    }
    return newNode;
}

void Print(ListNode *nd)
{
    while (nd)
    {
        if (nd->val < INT_MAX)
        {
            cout << nd->val << '\t';
        }
        nd = nd->next;
    }
}
```

```
ListNode* BuildList()
{
    ListNode *head = nullptr;
    ListNode *tail = nullptr;

    // 创建链表
    while (true)
    {
        int val;
        cin >> val;
        if (val < 0)
        {
            break;
        }

        tail = Insert(tail, val);
        if (head == nullptr)
        {
            head = tail;
        }
    }

    // 额外加一个虚拟节点, 使得
    // 之后要删除的结点不是尾节点
    Insert(tail, INT_MAX);
    return head;
}

int main()
{
    ListNode *head = BuildList();

    // 输出链表
    Print(head);

    // Release memory
    ListNode *curr = head;
    while (curr)
    {
        ListNode *nd = curr;
        curr = curr->next;
        delete nd;
    }

    return 0;
}
```


思考题

给你一个linked list，看有没有cycle？

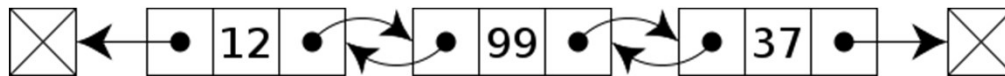
```
bool HasCycle(ListNode * pHead)
```

进一步，找出cycle的起点是哪个结点。

注意，要求空间复杂度 $O(1)$ ，也就是把不能保存整个链表来查询。

双向链表Doubly linked list

- `std::list<>`



```
struct ListNode
{
    int val;
    ListNode *prev;
    ListNode *next;
};

ListNode *head = new ListNode{12};
```

注意这里头结点初始化时，参数列表只给了`val`的值，`prev`和`next`设置缺省值`0`，即空指针

双向链表类声明

```
class DoublyList
{
private:
    ListNode * _head;
    ListNode * _tail;

    DoublyList(const DoublyList &);
    DoublyList & operator= (const DoublyList &);
public:
    DoublyList() : _head(nullptr), _tail(nullptr)
    {
    }

    ~DoublyList();

    void PushBack(int val);

    ListNode * Search(int val);

    void Insert(ListNode * node, int val);

    void Delete(ListNode * node);
};
```

```
DoublyList::~DoublyList()
{
    while (_head != nullptr)
    {
        ListNode * curr = _head;
        _head = curr->next;

        delete curr;
    }
}

void DoublyList::PushBack(int val)
{
    ListNode * newNode = new ListNode(val);

    if (_tail == nullptr)
    {
        _head = _tail = newNode;
    }
    else
    {
        _tail->next = newNode;
        newNode->prev = _tail;
        _tail = newNode;
    }
}
```

注意： copy constructor/assignment **private**

```
void print(DoublyList other)
{
}

void printref(const DoublyList & other)
{
}

int main()
{
    DoublyList dlList;
    dlList.PushBack(10);
    dlList.PushBack(20);
    dlList.PushBack(15);

    DoublyList copy(dlList); // Error: copy constructor is private
    DoublyList copy = dlList; // Error: assignment is private
    print(dlList); // Error: copy constructor is private
    printref(dlList); // OK

    return 0;
}
```

查找算法

- 从头节点开始，看下一个是不是要查找的节点。
- 时间复杂度 $O(n)$

```
ListNode * DoublyList::Search(int val)
{
    if (_head == nullptr)
    {
        return nullptr;
    }

    ListNode * curr = _head;
    while (curr != nullptr)
    {
        if (curr->val == val)
        {
            return curr;
        }

        curr = curr->next;
    }

    return nullptr;
}
```

插入算法

- 要插入一个节点，做法就是把当前节点指向新节点，然后新节点指向下一个节点，注意有两个指针prev/next
- 时间复杂度 $O(1)$ ，前提是知道要插入节点的地址

```
void DoublyList::Insert(ListNode * node, int val)
{
    ListNode * newNode = new ListNode(val);
    if (node == nullptr)
    {
        _head = _tail = newNode;
        return;
    }

    ListNode * nextNode = node->next;

    node->next = newNode;
    newNode->prev = node;
    newNode->next = nextNode;
    if (nextNode != nullptr)
    {
        nextNode->prev = newNode;
    }
    else
    {
        _tail = newNode;
    }
}
```

删除算法

- 只要修改前一个节点的后指针和后一个节点的前指针
- 时间复杂度 $O(1)$

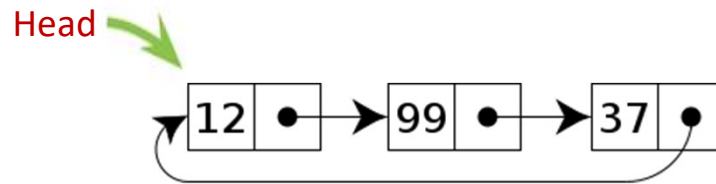
```
void DoublyList::Delete(ListNode * node)
{
    if (node == nullptr)
    {
        return;
    }

    ListNode * prevNode = node->prev;
    ListNode * nextNode = node->next;
    if (prevNode != nullptr)
    {
        prevNode->next = nextNode;
    }
    else
    {
        _head = nextNode;
    }

    if (nextNode != nullptr)
    {
        nextNode->prev = prevNode;
    }
    else
    {
        _tail = prevNode;
    }

    delete node;
}
```

循环链表Circular Linked List



```
struct ListNode
{
    int val;
    ListNode *next;
};
```

```
ListNode *head = new ListNode{12, nullptr};
```

注意这里尾节点的`next`不再指向`nullptr`，而是指向头指针

上机实验

- 给你doubly linked list，把它翻转过来

`void Reverse(ListNode * head)`

- 思考题：把单链表修改成循环链表
 - 建立循环链表比你想象的要容易
 - 在遍历循环链表时要注意死循环

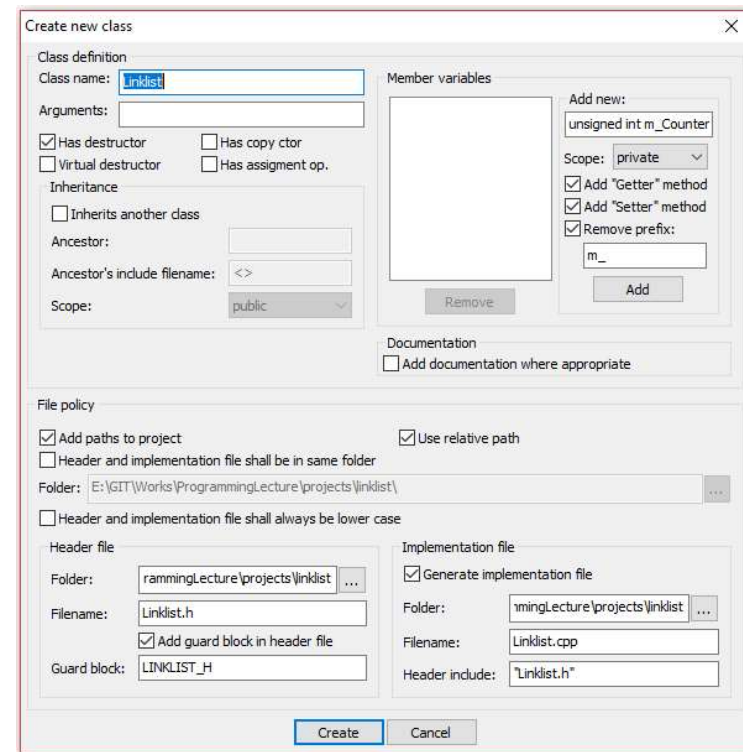
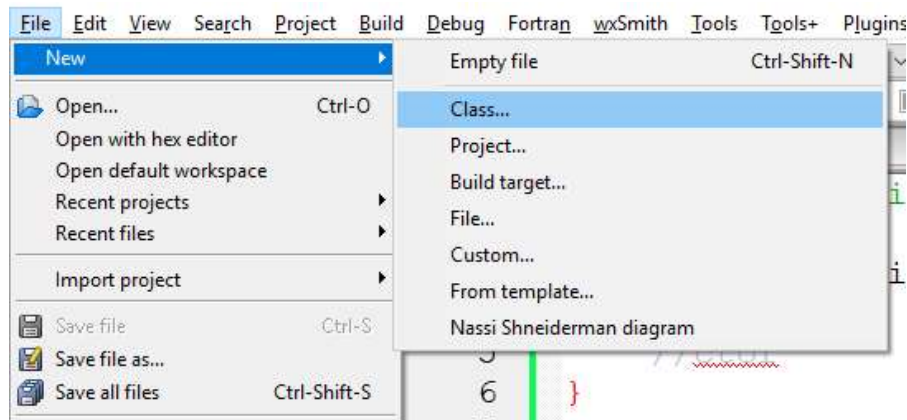
- **作业：**在现有的单链表类基础上实现单链表翻转功能

`void Reverse(ListNode * head)`

要求封装链表（同样适用于IntArray）

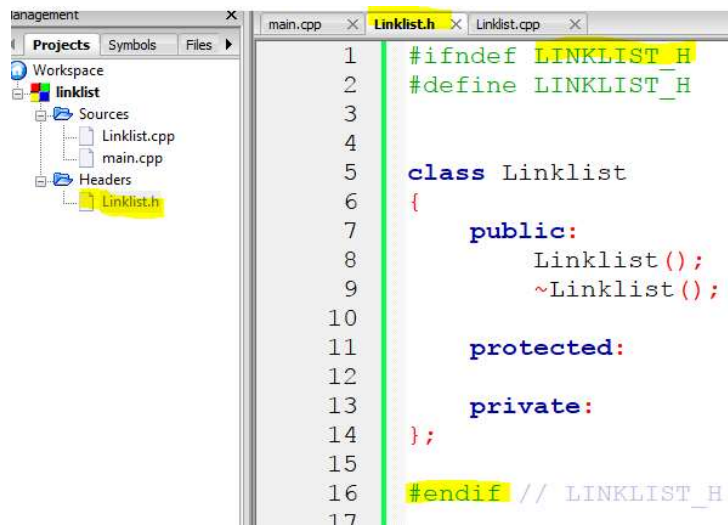
链表类

在项目里面添加类文件



链表类

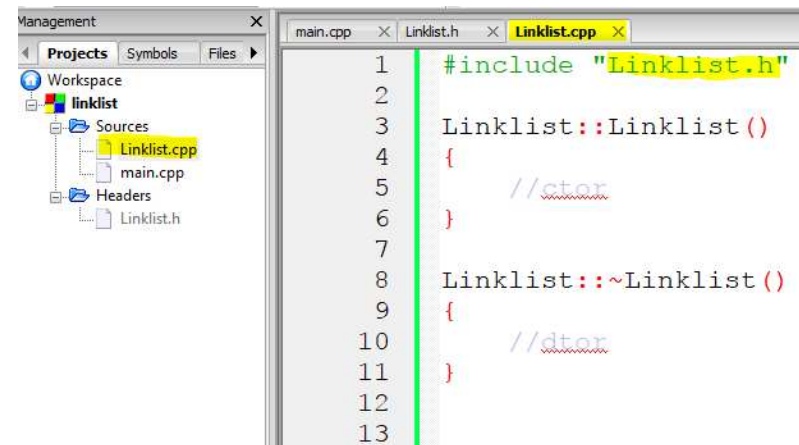
- 自动生成的头文件和代码文件



The screenshot shows an IDE window titled 'Management' with three tabs: 'main.cpp', 'Linklist.h', and 'Linklist.cpp'. The 'Linklist.h' tab is active, displaying the following code:

```
1  #ifndef LINKLIST_H
2  #define LINKLIST_H
3
4
5  class Linklist
6  {
7      public:
8          Linklist();
9          ~Linklist();
10
11      protected:
12
13      private:
14  };
15
16  #endif // LINKLIST_H
17
```

The left sidebar shows a project named 'linklist' with a 'Sources' folder containing 'Linklist.cpp' and 'main.cpp', and a 'Headers' folder containing 'Linklist.h'.



The screenshot shows the same IDE window with the 'Linklist.cpp' tab active, displaying the following code:

```
1  #include "Linklist.h"
2
3  Linklist::Linklist()
4  {
5      //ctor
6  }
7
8  Linklist::~~Linklist()
9  {
10     //dctor
11 }
12
13
```

The left sidebar shows the same project structure, but the 'Linklist.h' file is now in the 'Headers' folder.

使用链表类

Main.cpp

```
#include "Linklist.h"

int main()
{
    Linklist list(9);
    Node *nd = list.Search(5);
    if (nd)
    {
        list.Insert(nd, 10);
    }
    list.Delete(nd);
    list.Print();

    return 0;
}
```

Linklist.h

```
#ifndef LINKLIST_H
#define LINKLIST_H
#include <iostream>

struct Node
{
    int key;
    Node *next;
};

class Linklist
{
public:
    Linklist(int num);
    ~Linklist();

    Node * Search(int val);
    Node * Insert(Node *node, int val);
    void Delete(Node *node);

    void Reverse(Node *head);

    friend std::ostream & operator<< (std::ostream &out, const Linklist &list);

private:
    Linklist(const Linklist &other); // private copy constructor
    Linklist& operator=(const Linklist &rhs); // private assignment overload

    Node *m_pHead;
};
#endif // LINKLIST_H
```

```
link list:
1      2      3      4      10      6      7      8      9
```

使用IntArray类

Main.cpp

```
#include "intarray.h"
int main()
{
    IntArray myarray(10);
    myarray.Insert(0, 23);
    myarray.Insert(0, 12);
    myarray.Delete(1);

    return 0;
}
```

IntArray.h

```
#ifndef INTARRAY_H
#define INTARRAY_H
#include <iostream>
#include <cassert>
using namespace std;

class IntArray
{
public:
    IntArray(int size);

    // virtual destructor: SortedIntArray inherits
    virtual ~IntArray();
    // copy constructor
    IntArray(const IntArray & other);
    // assignment overload
    IntArray & operator= (const IntArray & rhs);

    void Insert(int index, int value);
    void Delete(int index);

    inline int Length() const { return _size; }

    void Reserve(int capacity);

    int & operator[] (int index);
    const int & operator[] (int index) const;

    friend ostream & operator<< (ostream & out, const IntArray & list);

private:
    int * _pData;
    int _capacity;
    int _size;

    void GetArray(int capacity);
    void Reset();
};
#endif // INTARRAY_H
```

reference

Comparison of list data structures						
	Linked list	Array	Dynamic array	Balanced tree	Random access list	hashed array tree
Indexing	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$ ^[4]	$\Theta(1)$
Insert/delete at beginning	$\Theta(1)$	N/A	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(n)$
Insert/delete at end	$\Theta(1)$ when last element is known; $\Theta(n)$ when last element is unknown	N/A	$\Theta(1)$ amortized	$\Theta(\log n)$	$\Theta(\log n)$ updating	$\Theta(1)$ amortized
Insert/delete in middle	search time + $\Theta(1)$ ^{[5][6][7]}	N/A	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$ updating	$\Theta(n)$
Wasted space (average)	$\Theta(n)$	0	$\Theta(n)$ ^[8]	$\Theta(n)$	$\Theta(n)$	$\Theta(\sqrt{n})$

Q&A

Thanks!