# 数据结构与算法

# DATA STRUCTURE

第六讲 动态数组和链表

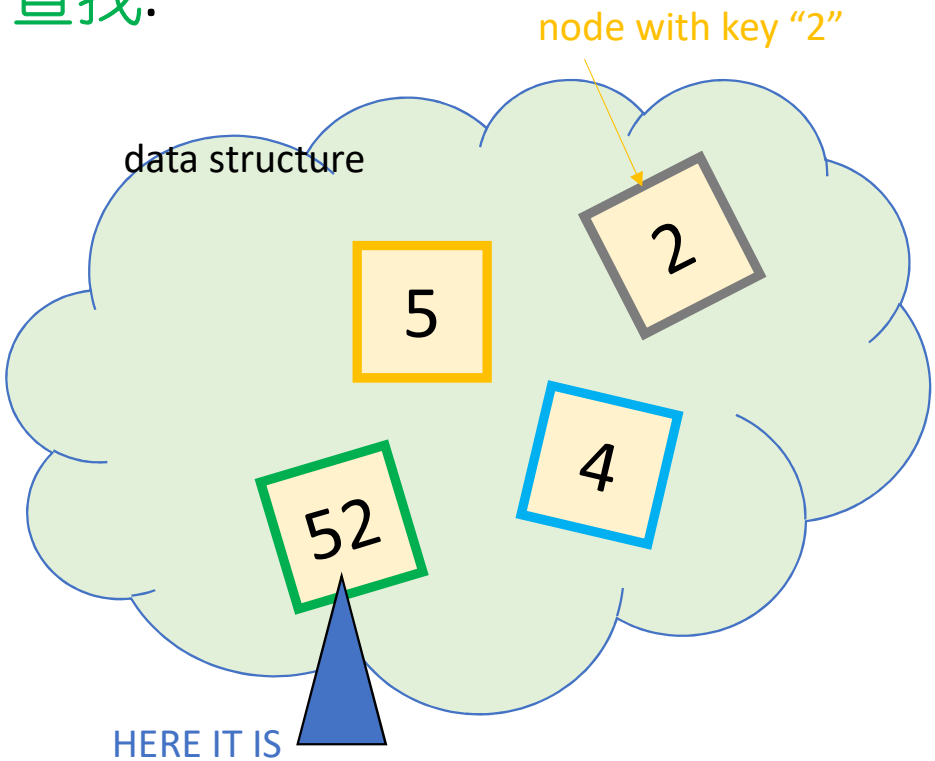
信息管理与工程学院

2017 - 2018 第一学期

# 数据结构的目的:

- 维护一个动态的数据元素存储方式,
- 使得支持快速的插入/删除/查找.

- INSERT 5
- DELETE 4
- SEARCH 52



# 数据结构核心问题

1. 存储数据方式，空间复杂度
2. 查找search，时间复杂度
3. 插入insert，时间复杂度
4. 删除delete，时间复杂度

# 基本数据结构

- 数组，区别C++ Array，std::Array
- 链表

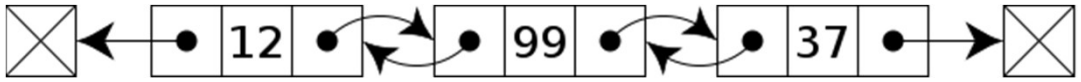
# 静态结构

- std::Array，C++ array是静态结构，
- 不能insert/delete，或者说只有暴力的方法
- 链表一般来说是动态的

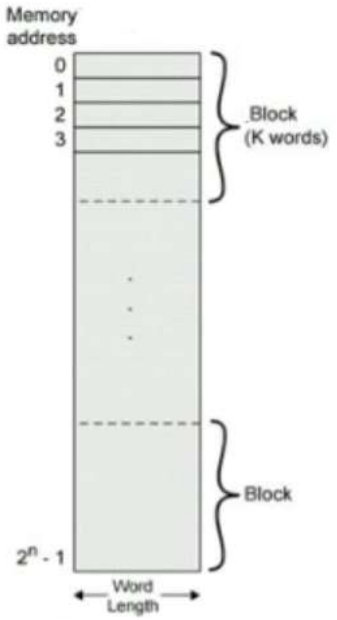
# 动态结构：插入/删除/查找

- std::vector 动态数组，按dynamic array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doubly linked list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）



# 数据的局部特性Data locality



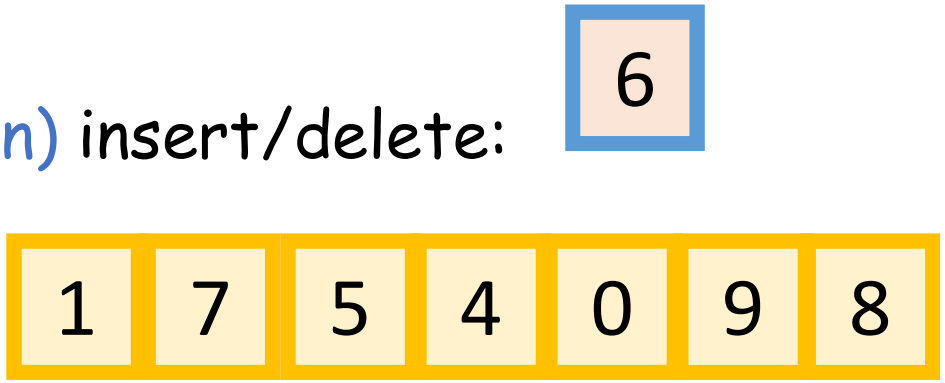

- 属于计算机里一种可以预测的行为
  - 一次IO读取的利用率，block size
  - 缓存caching，包括时间locality，空间locality
  - 预取操作prefetching
  - CPU分支预测
- 大数据上一般用数据块读取次数来衡量时间复杂度，而不是单个数据节点的操作次数。
- 研究数据的存放方式来增加data locality是一个研究方向

# 主要内容

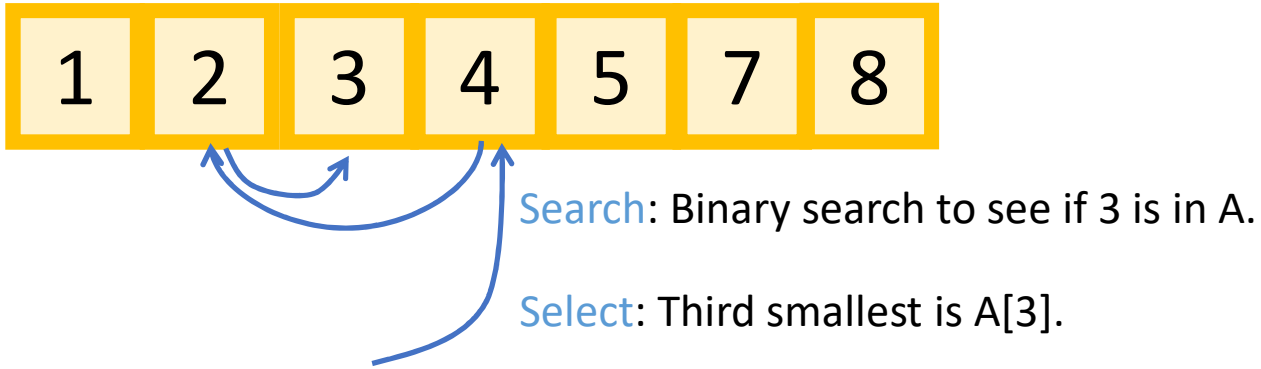
- 动态数组
  - std::vector的底层结构
  - 不讲sequential list，和动态数组差不多，后面数据结构不适用
- 双链表
  - std::list的底层结构
  - 简单讲解单链表

# 动态数组Dynamic Arrays

- $O(n)$ insert/delete:




- SortedArray$O(\log(n))$ search, $O(1)$ select:

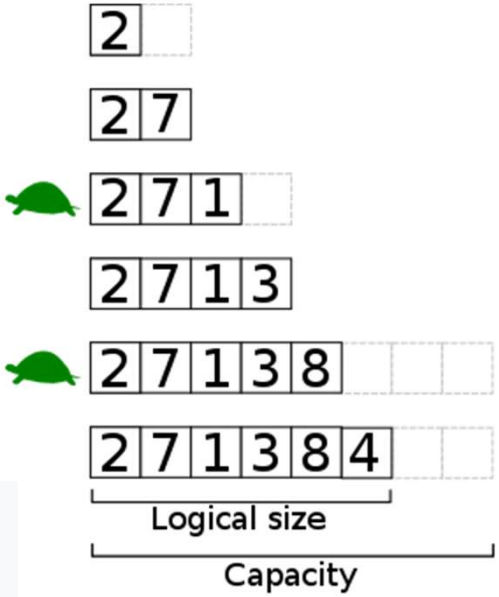


# 动态数组Dynamic array

- 前面IntArray其实是静态结构，size固定的
- 要支持insert/delete
- 最简单的做法是插入一个元素就重新分配一次内存

# 插入算法

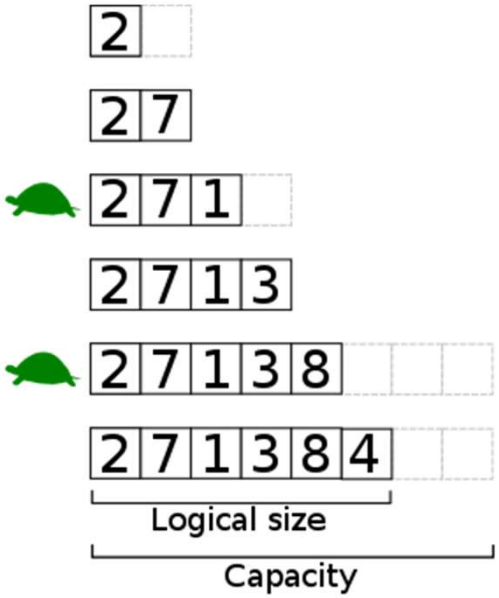
- 引入另外一个私有变量capacity，区别于size
- 算法是在数组size达到capacity时候，重新分配2倍大小的新数组
- 把旧数组元素复制到新数组
- 正常情况是把要插入的位置后面的数据元素往后移动一个位置，然后再插入




```
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

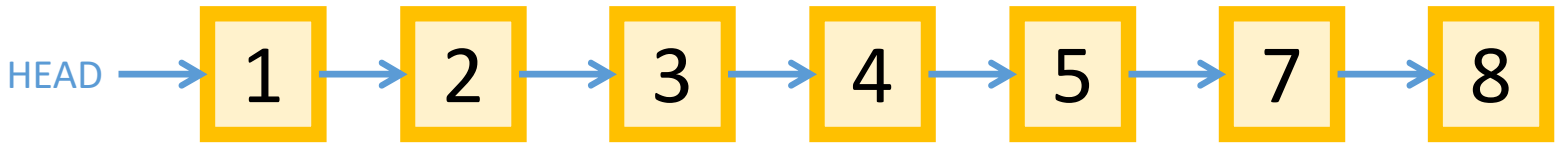
- 在IntArray基础上实现insert/delete算法
  - Void insert(int index, int value);
  - Void delete(int index);

- 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

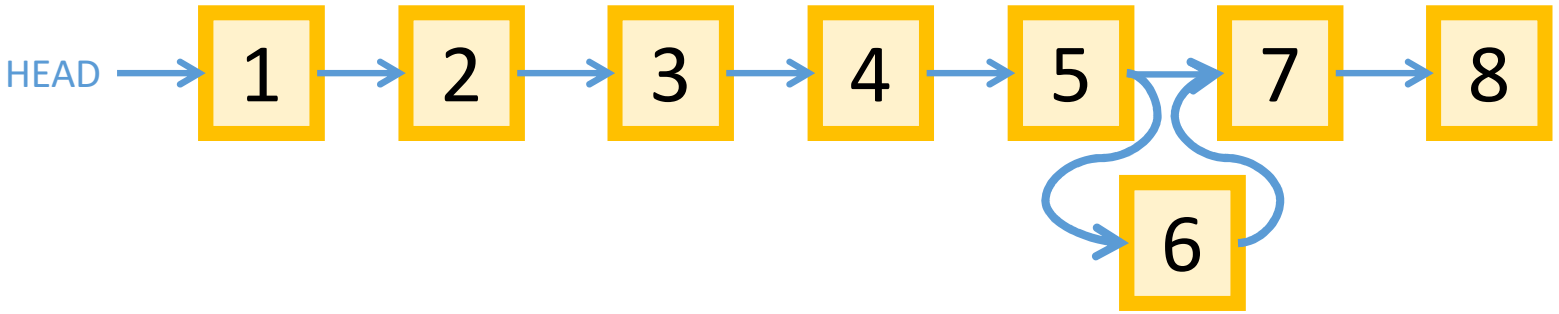
- 一串链接起来的数据节点
- 在内存里是离散分布
- 每个节点除了数据，还有下一个节点的地址
- 插入/删除复杂度只有O（1）
- 但是查找复杂度是O（n）

# Linked list

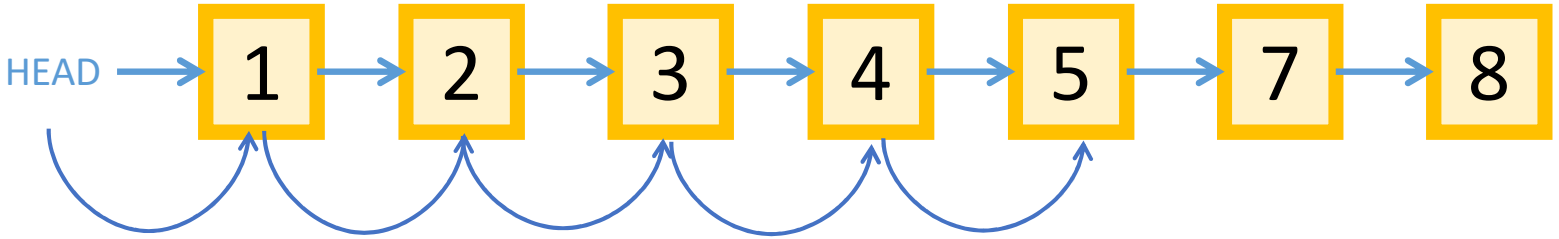



# Linked list

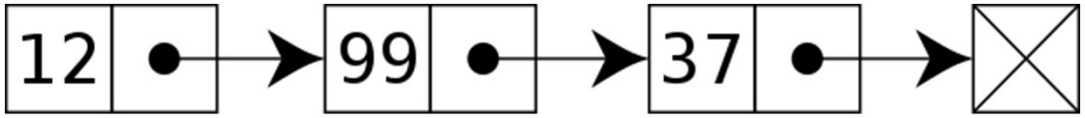
- O(1) insert/delete (assuming we have a pointer to the location of the insert/delete):




- O(n) search:



# 单链表Singly linked list




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

# 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 * Search(ListNode * head, 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
- 时间复杂度O(1)

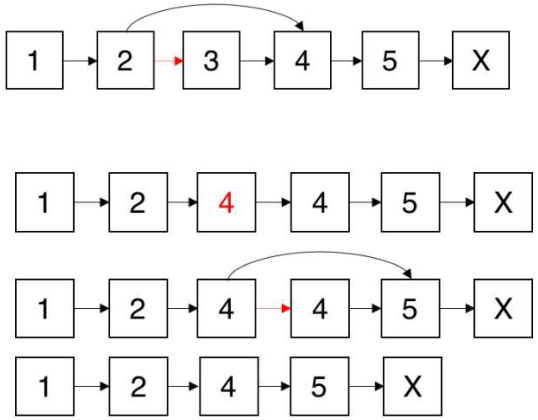
```
void insert(ListNode * node, int val)
{
    if (node == nullptr)
    {
        return;
    }

    ListNode * newNode = new ListNode(val);

    ListNode * nextNode = node->next;
    node->next = newNode;
    newNode->next = nextNode;
}
```

# 删除算法

- 要删除一个节点，一般的算法是要找到前一个节点，然后让前一个节点指向后一个节点
- 改进的做法是不用查找前一个节点，除非删除的是最后一个节点




```
void delete(ListNode * node)
{
    if (node == nullptr)
    {
        return;
    }

    ListNode * nextNode = node->next;
    if (nextNode == nullptr)
    {
        /*TODO: delete last node */
        return;
    }

    node->val = nextNode->val;
    node->next = nextNode->next;

    delete nextNode;
}
```

# 思考题

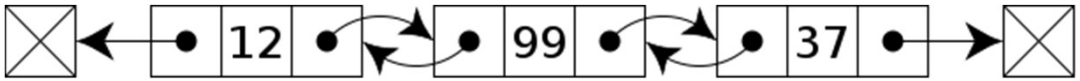
给你一个linked list，看有没有cycle？
bool HasCycle(ListNode * head)

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(int x) : val(x), prev(nullptr), next(nullptr) {}
};
```

# 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)

```
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)

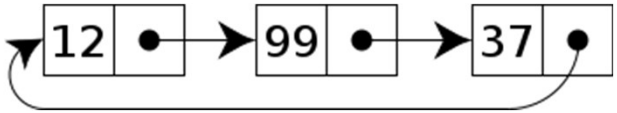
```
void DoublyList::Delete(ListNode * node)
{
    if (node == nullptr)
    {
        return;
    }

    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



# 上机实验

- 给你doubly linked list，把它翻转过来
- Void reverse(ListNode * head)

- 作业：把单链表翻转过来

# reference

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

Q&A

Thanks!