C,C++ Data structure and Algorithm

Table of Contents

[I. Cấu trúc dữ liệu 1](#_Toc475342405)

[1. Cấu trúc danh sách 1](#_Toc475342406)

[2. Stack - Ngăn xếp 18](#_Toc475342407)

[3. Queue – Hàng đợi 22](#_Toc475342408)

[4. Cấu trúc cây 25](#_Toc475342409)

[5. Bảng băm 83](#_Toc475342410)

[6. Heap 98](#_Toc475342411)

[7. Đồ thị 104](#_Toc475342412)

[II. Các thuật toán 106](#_Toc475342413)

[1. Tính hiệu quả của giải thuật 106](#_Toc475342414)

[2. Giải thuật đệ qui 109](#_Toc475342415)

[3. Các giải thuật sắp xếp 118](#_Toc475342416)

[4. Thuật toán tìm đường A\* 141](#_Toc475342417)

[5. Thuật toán nén 150](#_Toc475342418)

[6. Triển khai Ma trận với STL 150](#_Toc475342419)

[7. Thư viện template chuẩn STL 153](#_Toc475342420)

## Cấu trúc dữ liệu

### Cấu trúc danh sách

Danh sách liên kết là 1 cấu trúc dữ liệu cơ bản, được sử dụng để khắc phục hạn chế của mảng (cố định về kích thước). C++ nói chung và cụ thể là thư viện STL đã cung cấp sẵn một kiểu dữ liệu List. Tuy nhiên tôi vẫn muốn chia sẻ bài viết này để nêu rõ về bản chất của danh sách liên kết và một số thao tác cơ bản trên nó.

1. Danh sách là mảng đơn giản

Một danh sách có thể là 1 mảng thuần túy của C++, tạo ra với bộ chứa cố định, hoặc std::vector, có thể mở rộng vùng chứa khi cần.

Std::vector là phiên bản hoàn hảo của mảng cấp phát động, vector giải quyết được vấn đề mở rộng bộ chứa phần tử, không bắt buộc người dùng phải xác định số phần tử tối đa để cấp phát.

Std::vector cho phép in danh sách phần tử theo thời gian tuyến tính, thời gian cho thao tác tìm kiếm cố định O(1). Tuy nhiên, thao tác chèn và xóa tốn chi phí, tùy vị trí, trường hợp xấu nhất là rơi vào phần tử đầu tiên, mất O(N).

Trường hợp tốt nhất là rơi vào cuối list: O(1).

Chúng ta sử dụng cấu trúc mảng vector khi chỉ cần chèn nhanh phần tử vào cuối danh sách, cần thao tác truy xuất nhanh.

Tuy nhiên, nếu tồn tại yêu cầu chèn, xóa trên toàn danh sách thì nên dùng cấu trúc danh sách liên kết List.

1. Triển khai template vector

Vector là kiểu đối tượng đầu tiên, không giống như kiểu mảng thuần túy trong C++, vector được quá tải sao chép, heap memory được thu hồi qua hàm hủy.

Những đặc tính của kiểu mảng trong C++ cần nhớ:

* Mảng đơn giản là một con trỏ trỏ tới một vùng nhớ, kích thước mảng phải được duy trì bởi lập trình viên.
* Vùng nhớ cấp phát bởi new[] , phải thu thồi qua delete[].
* Vùng nhớ có thể mở rộng với new(Vùng nhớ lớn hơn được cấp và khởi tạo giá trị với vùng nhớ cũ, sau đó vùng nhớ cũ bị thu hồi).

Tránh nhầm lẫn với lớp thư viện, chúng ta đặt tên triển khai mới là template Vector.

Chúng ta điểm qua các ý chính:

1.Vector duy trì một mảng cơ bản (qua 1 con trỏ tới vùng nhớ), capactiy của nó, và số phần tử.

2. Vector triển khai Big-5: Hỗ tợ deep-copy với Copy Constructor và Copy Assignment operator=, hỗ trợ C++11 Move semantic với Move Constructor, Move Assignment operator=. Destructor sẽ thu thồi vùng nhớ cho mảng.

3. Hỗ trợ phương thức resize, thay đổi kích thước cho Vector và phương thức reserve, thay đổi capacity cho Vector. Reserve sẽ cấp phát vùng nhớ mới, sao chép giá trị vùng nhớ cũ sang, sau đó thu hồi.

4. Triển khai phiên bản Accessor và Mutator cho quá tải toán tử truy xuất phần tử ngẫu nhiên [ ].

5. Các phương thức cơ bản: size, empty, clear, back, pop\_back và push\_back. Riêng push\_back có thể gọi reserve khi size = capacity.

6. Hỗ trợ kiểu nội bộ iterator và const\_iterator, cùng phương thức begin – end.

Như một bản sao của Vector STL, nhưng khả năng kiểm soát lỗi sẽ đề cập sau.

#include <algorithm>

template <typename Object>

class Vector

{

public:

/\*Hàm dựng nhận tham số initSize,mặc định zero, khởi tạo mảng thành viên có kích thước capacity lớn hơn initSize 1 chút. Vector có thể push\_back mà không cần thay đổi capacity\*/

explicit Vector( int initSize = 0 ) : theSize{ initSize },theCapacity{ initSize + SPARE\_CAPACITY }

{ objects = new Object[ theCapacity ]; }

/\*The Hàm dựng sao chép mặc định, điều chỉnh capacity sau đó sao chép giá trị từng phần tử\*/

Vector( const Vector & rhs ) : theSize{ rhs.theSize },theCapacity{ rhs.theCapacity }, objects{ nullptr }{

objects = new Object[ theCapacity ];

for( int k = 0; k < theSize; ++k )

objects[ k ] = rhs.objects[ k ];

}

/\*Quá tải phép gán sao chép mặc định, dùng std::swap để move đối tượng tạm copy sang \*this, yêu cầu vector phải triển khai move assignment và move constructor\*/

Vector & operator= ( const Vector & rhs ){

Vector copy = rhs;

std::swap( \*this, copy );

return \*this;

}/\* Phương thức chưa tốt, đặc biệt khi cả 2 vector cùng size, có thể kiểm tra size và tốt hơn là sao chép từng phần tử dùng phép gán = của Object\*/

~Vector( ){ delete [ ] objects; }

/\*move constructor\*/

Vector( Vector && rhs ) : theSize{ rhs.theSize },theCapacity{ rhs.theCapacity }, objects{ rhs.objects } {

rhs.objects = nullptr;

rhs.theSize = 0;

rhs.theCapacity = 0;

}

/\*move assignment\*/

Vector & operator= ( Vector && rhs ) {

std::swap( theSize, rhs.theSize );

std::swap( theCapacity, rhs.theCapacity );

std::swap( objects, rhs.objects );

return \*this;

}

/\* Phương thức resize chỉ thay đổi giá trị dữ liệu thành viên theSize, nếu newSize > theCapacity, phải gọi reserve để mở rộng capacity của Vector(\*2 lần + 1). Chi phí mở rộng là khá đắt.\*/

void resize( int newSize ) {

if( newSize > theCapacity )

reserve( newSize \* 2 );

theSize = newSize;

}

/\*Phương thức reserve để mở rộng capacity cho Vector\*/

void reserve( int newCapacity ) {

/\*Reserve cũng có thể dùng để giảm capacity, giảm kích thước mảng lại, nhưng chỉ khi newCapacity >= TheSize. Nếu không nó được bỏ qua, return\*/

if( newCapacity < theSize )

return;

/\* Đầu tiên nó cấp phát mảng mới với newCapacity\*/

Object \*newArray = new Object[ newCapacity ];

/\* Sau đó move từng phần tử mảng cũ sang mảng mới dùng std::move\*/

for( int k = 0; k < theSize; ++k )

newArray[ k ] = std::move( objects[ k ] );

theCapacity = newCapacity;

/\*move mảng mới vào mảng cũ\*/

std::swap( objects, newArray );

/\*Sau cùng thu hồi vùng nhớ cũ\*/

delete [ ] newArray; }

/\*Tiếp theo là Accessor và Mutator của quá tải toán tử [].Kiểm soát lỗi đơn giản là kiểm tra index phải từ 0 tới size()-1, và throw exception nếu không đúng\*/

Object & operator[]( int index ){ return objects[ index ]; }

const Object & operator[]( int index ) const{ return objects[ index ]; }

/\*empty, size, capacity triển khai đơn giản như bên dưới\*/

bool empty( ) const{ return size( ) == 0; }

int size( ) const{ return theSize; }

int capacity( ) const{ return theCapacity; }

/\*push\_back kiểm tra size và capacity, mở rộng nếu size == capacity.

Toán tử hậu tố ++ được dùng để tăng theSize sau khi gán giá trị\*/

void push\_back( const Object & x ){

if( theSize == theCapacity )

reserve( 2 \* theCapacity + 1 );

objects[ theSize++ ] = x;

}

void push\_back( Object && x ){

if( theSize == theCapacity )

reserve( 2 \* theCapacity + 1 );

objects[ theSize++ ] = std::move( x );

}

/\*Cùng ý tưởng với objects[ theSize++ ] và objects[ theSize++ ] khi thảo luận iterators:

\*itr++ truy xuất phần tử trước, rồi tăng itr.

\*++itr tăng itr trước, rồi mới truy xuất phần tử\*/

/\*pop\_back và back có thể throw exception nếu size = 0\*/

void pop\_back( ){--theSize;}

const Object & back ( ) const{return objects[ theSize - 1 ];}

/\*iterator và const\_iterator, cùng 4 phương thức begin/end tương ứng\*/

typedef Object\* iterator;

typedef const Object\* const\_iterator;

iterator begin( ){ return &objects[ 0 ]; }

const\_iterator begin( ) const{ return &objects[ 0 ]; }

iterator end( ){ return &objects[ size( ) ]; }

const\_iterator end( ) const{ return &objects[ size( ) ]; }

static const int SPARE\_CAPACITY = 16;

/\*Dữ liệu thành viên gồm: theSize, theCapacity và mảng đối tượng object\*/

private:

int theSize;

int theCapacity;

Object \* objects;

};

Như vậy, STL iterator được thiết kế như con trỏ tới đối tượng, do đặc tính của con trỏ trong C,C++, khi chúng ta dùng toán tử nhất nguyên ++, iterator sẽ trỏ tới phần tử tiếp theo trong mảng.

Bản thân iterator cũng mang địa chỉ của phần tử, begin/end trả về địa chỉ phần tử đầu tiên và cuối cùng trong mảng.

Sử dụng Vector sẽ tăng thêm chút chi phí so với việc dùng mảng trong C++.

Tuy nhiên, do không có khả năng kiểm soát lỗi, iterator có thể đi qua phần tử cuối, ++itr hay \*itr phải phát ra lỗi.

Để làm được, iterator và const\_iterator phải triển khai thành lớp lồng thực sự bên trong Vector, hơn là con trỏ thuần.

Việc triển khai lớp lồng là khá phổ biến, như trong triển khai List phần tiếp theo.

1. Danh sách liên kết đơn
2. Đặc tính

DSLK được triển khai để tránh chi phí phát sinh tuyến tính do việc chèn, xóa khi các phần tử được nạp không phải dạng liên tiếp.

Để indanh sách hay tìm, chúng ta phải duyệt từ đầu tới cuối node dùng liên kết \*next, thời gian tăng tuyến tính theo số lượng phần tử. Nói chung sẽ không hiệu quả như cấu trúc mảng thuần.

Trên thực tế, trong danh sách đã sắp xếp, việc tìm kiếm thường xuyên các phần tử như findKth(2), findKth(3), findKth(4), và findKth(6) có thể làm trong 1 lần duyệt.

Thao tác Xóa có thể thực hiện trong 1 lần đổi liên kết.

Thao tác chèn chỉ dùng 1 lần gọi new, sau đó điều chỉnh liên kết \*next.

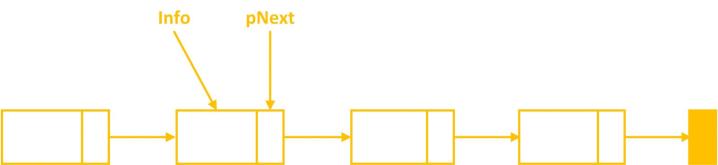
Ưu điểm là nếu biết vị trí để xóa hay chèn, chúng ta sẽ không cần phải duyệt nhiều. Trường hợp đặc biệt thêm hay xóa phần tử đầu/cuối, chỉ mất O(1), nếu duy trì node đầu và cuối.

Ý tưởng về đường liên kết thứ 3 tới node cuối cùng là không cần thiết, tốn chi phí để duy trì khi xóa hay chèn.

1. Cấu trúc

Cũng giống như mảng, danh sách liên kết cũng bao gồm các phần tử, có mối liên hệ với nhau. Tôi gọi mỗi phần tử đó là một Node. Node được xem là trái tim của danh sách liên kết, mỗi Node sẽ lưu trữ 2 thông tin:

* Thông tin dữ liệu: Lưu trữ các thông tìn về chính Node đó.
* Thông tin liên kết: Lưu trữ địa chỉ của phần tử kế tiếp trong danh sách, hoặc lưu trữ giá trị NULL nếu phần tử đó nằm cuối danh sách.

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_4/106/ss_1.jpg)

Một cách tổng quát ta có:

1. struct SNode{
2. Data Info;
3. SNode\* pNext;
4. };

Mỗi phần tử trong trong danh sách liên kết đơn là một biến động sẽ được yêu cầu cấp phát khi cần thiết, danh sách liên kết đơn chính là sự liên kết các biến này với nhau do đó ta hoàn toàn chủ động về số lượng các phần tử.

Để đơn giản, trong bài viết này tôi sẽ lấy ví dụ danh sách liên kết đơn lưu trữ các số nguyên.

Node của danh sách liên kết sẽ được định nghĩa như sau:

1. struct SNode
2. {
3. int Data;
4. SNode\* pNext;
5. };

 Tôi sử dụng một phương thức GetNode để cấp phát động một Node khi cần thiết:

1. SNode\* GetNode(int x)
2. {
3. SNode \*p = new SNode;
5. p->Data = x;
6. p->pNext = NULL;
8. return p;
9. }

Bây giờ chúng ta bắt đầu tìm hiểu một số phương thức đơn giản tạo nên sự liên kết các Node để tạo ra một danh sách liên kết đơn hoàn chỉnh.

1. Một số thao tác cơ bản trên danh sách liên kết đơn
2. Tổ chức

Trong danh sách liên kết đơn, các Node sẽ không được lưu liên tiếp nhau trên bộ nhớ, Node trước sẽ mang thông tin địa chỉ của Node sau, như vậy nếu bạn xử lý lỗi một Node sẽ dẫn đến tính huống xấu nhất, ta sẽ mất toàn bộ thông tin của các Node phía sau.

Chúng ta sẽ bắt đầu làm việc với các thao tác cơ bản trên một danh sách liên kết đơn. Giả sử tôi có định nghĩa sau:

1. struct SNode{
2. int Data;
3. SNode\* pNext;
4. };
6. struct SList{
7. SNode\* pHead;
8. SNode\* pTail;
10. SList(){}
11. SList(SNode\* Head, SNode\* Tail) {
12. this->pHead = Head;
13. this->pTail = Tail;
14. }
15. };

Nếu biết được địa chỉ đầu tiên trong danh sách liên kết ta có thể dựa vào thông tin pNext để truy xuất đến các phần tử còn lại, do đó ta sẽ dùng một con trỏ pHead để lưu lại địa chỉ Node đầu tiên của danh sách. Trong một số trường hợp ta cũng cần thao tác trên phần tử cuối cùng của danh sách, nên tôi dùng thêm một con trỏ pTail để lưu trữ địa chỉ của Node cuối cùng trong danh sách.

1. Chèn vào đầu danh sách

Như đã trình bày ở trên, khi thao tác với mỗi Node trên danh sách liên kết ta cần thực hiện cẩn thận, đúng thứ tự để tránh mất thông tin của các Node phía sau. Dưới đây là thứ tự các bước chèn một phần tử vào đầu mảng.

**Bước 1:** cấp phát một Node mới (new\_element).

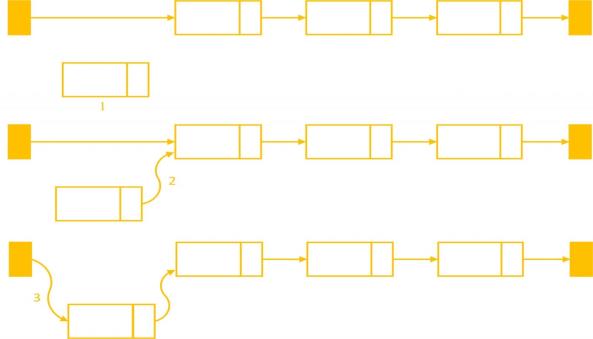
**Bước 2:** gán pNext của Node mới trỏ đến Node đầu (cũ).

1. new\_element->pNext = pHead;

**Bước 3:** cập nhập lại giá trị pHead

1. pHead = new\_element;

Lưu đồ bên dưới thể hiện từng bước để thêm vào một Node mới ở đầu danh sách

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_4/106/ss_2.jpg)

1. Chèn vào cuối danh sách

Tương tự với thao tác chèn vào đầu danh sách, chèn vào cuối danh sách chúng ta chỉ cần cập nhập lại con trỏ pTail.

**Bước 1:**cấp phát một Node mới (new\_element).

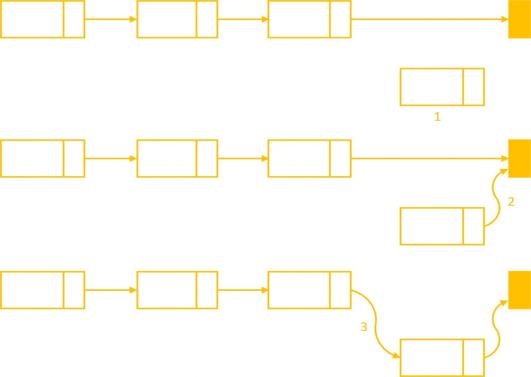
**Bước 2:** gán lại pNext của Node cuối cùng (cũ) sẽ trỏ đến Node mới.

1. pTail->pNext = new\_element;

**Bước 3:** cập nhập lại giá trị pTail, bây giờ Node cuối của danh sách là Node chúng ta vừa thêm.

1. pTail = new\_element;

Đây là lưu đồ thứ tự thực hiên các bước ở trên

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_4/106/ss_3.jpg)

1. Chèn vào danh sách sau một phần tử q

Chèn vào danh sách sau một phần tử q nào đó, chèn vào giữa danh sách không cần cập nhập lại hai con trỏ pHead và pTail tuy nhiên chúng ta cần hết sức cần thận để tránh mất dữ liệu phía sau.

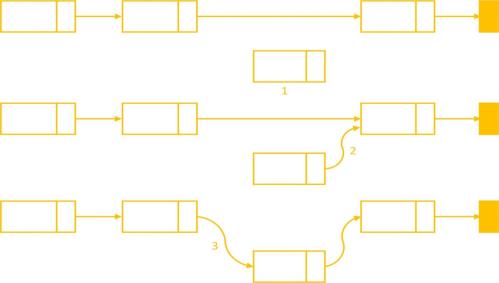
**Bước 1:** cấp phát một Node mới - new\_element.

**Bước 2:** gán pNext của Node mới bằng địa chỉ của Node sau q.

1. new\_element ->pNext = q-> pNext;

**Bước 3:** cập nhập lại pNext của Node q trỏ đến Node mới.

1. q-> pNext = new\_element;

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_4/106/ss_4.jpg)

1. Xóa một phần tử đầu danh sách

Xóa 1 phần tử ở đầu danh sách không chỉ đơn giản là cập nhập lại biến con trỏ pHead, mà ta phải giải phóng được vùng nhớ của Node cần xóa.

**Bước 1:** Khai báo một con trỏ p để lưu lại địa chỉ của Node đầu tiên.

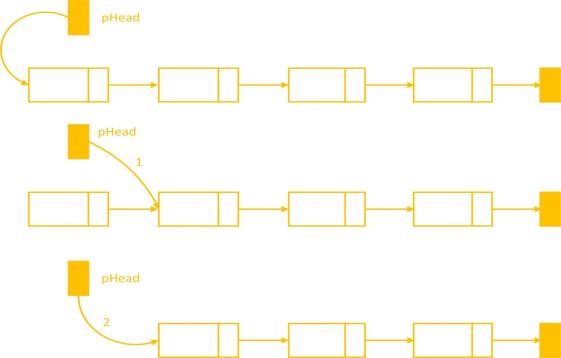
1. SNode\* p = q->pNext;

**Bước 2:** cập nhập lại giá trị pNext của Node q.

1. q->pNext = p->pNext;

**Bước 3:** giải phóng vùng nhớ của Node cần xóa.

1. delete p;

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_4/106/ss_5.jpg)

1. Xóa một phần tử đứng sau phần tử q

Tương tự như thao tác xóa phần tử đầu danh sách, để xóa một phần tử đứng sau phần tử q, ta thực hiện các bước sau:

**Bước 1:** Khai báo một con trỏ p để lưu lại địa chỉ của Node đầu tiên.

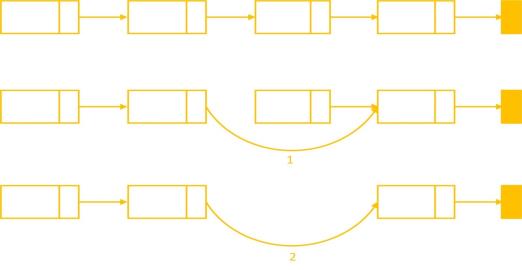
1. SNode\* p = pHead;

**Bước 2:** cập nhập lại giá trị của pHead.

1. pHead = pHead->pNext;

**Bước 3:** giải phóng vùng nhớ của Node cần xóa.

1. delete p;

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_4/106/ss_6.jpg)

1. Duyệt danh sách

Khi có được giá trị của pHead ta có thể dựa và thông tin pNext để duyệt lần lượt các phần tử của danh sách.

1. SNode\* p;
3. p = list.pHead;
4. while (p != NULL)
5. {
6. // Process Node
7. p = p->pNext;
8. }
9. Triển khai template List

Lớp template của chúng ta đặt tên là List, nhằm phân biệt với std::list, và là DSLK kép. Như đã đề cập trong phần triển khai lớp template Vector, iterator sẽ được triển khai thành lớp lồng trong List, thiết kế sẽ có 3 lớp và 1 cấu trúc:

* Lớp List: duy trì node đầu/cuối, kích thước, các phương thức khác.
* Cấu trúc Node: là một **private** nested struct, chứa liên kết và dữ liệu.
* Lớp const\_iterator: **public** nested class, chứa con trỏ tới node nó đại diện, triển khai đầy đủ các phương thức trên iterator, đa năng các toán tử như =, ==, !=, và ++. Toán tử \* sẽ được quá tải trả về const reference.
* Lớp iterator: triển khai tương tự như const\_iterator, nhưng toán tử \* trả về tham chiếu non-const.

Do đặc tính của C++, đối tượng iterator có thể dùng trên mọi phương thức cần const\_iterator, nhưng ngược lại thì không.

Iterator đánh dấu kết thúc sẽ được tạo ra như một node ở cuối danh sách. Ngoài ra, chúng ta có thể tạo thêm 1 node đầu danh sách, đánh dấu là iterator bắt đầu.

Những node thêm này được biết như node lính canh, node đầu gọi là header node, node cuối gọi là tail node.

Việc duy trì những node đặc biệt này giúp ích rất nhiều khi xử lí nhiều trường hợp phức tạp. Ví dụ, khi cần xóa node, sẽ phải xử lí theo cách khác nếu node đó là header hay tail node.

1. Private Struct Node

Node không được khai báo như một private nested class.

*Struct hiện diện như một thánh tích của C trong C++, dẫu vậy bản chất của struct đã theo OOP hơn rất nhiều.*

*Các dữ liệu, phương thức thành viên đều mặc định mang tính public nên rõ ràng không nhất thiết phải dùng từ khóa struct. Nhưng có điều thú vị là, việc wrap hầu hết các dữ liệu truy xuất trực tiếp bên trong struct lại rất phổ biến, mà không áp dụng cho phương thức.*

Việc triển khai private class cho Node là không cần thiết, bản thân Node đã nằm trong phạm vi private của List, không thể truy xuất từ bên ngoài được.

template<typename Object>

struct Node {

/\*Tất cả thành viên public\*/

Object data; //item

Node \*prev; //liên kết trước

Node \*next; //liên kết sau

/\*Hàm dựng copy\*/

Node( const Object & d = Object{ }, Node \* p = nullptr, Node \* n = nullptr ):

data{ d }, prev{ p }, next{ n } { }

/\*Hàm dựng MOVE\*/

Node(Object && d, Node \* p = nullptr, Node \* n = nullptr ):

data{ std::move( d ) }, prev{ p }, next{ n } { }

};

1. class const\_iterator

Const\_iterator đa năng cho các toán tử ==; !=; \* , toán tử ++ triển khai 2 phiên bản tiền tố và hậu tố, có kiểu trả về khác nhau.

Tùy theo trường hợp chúng ta sẽ dùng tiền tố hay hậu tố.

template<typename Object>

class const\_iterator{

public:

const\_iterator( ) : current{ nullptr } { }

//Quá tải \* trả về tham chiếu hằng

const Object & operator\* ( ) const { return retrieve( ); }

//Quá tải tiền tố ++

const\_iterator & operator++ ( ) {

current = current->next;

return \*this;

}

//Quá tải hậu tố ++

const\_iterator operator++ ( int ) {

const\_iterator old = \*this;

++( \*this );

return old;

}

//Quá tải so sánh

bool operator== ( const const\_iterator & rhs ) const { return current == rhs.current; }

bool operator!= ( const const\_iterator & rhs ) const { return !( \*this == rhs ); }

Chúng ta cho phép khai báo iterator để sử dụng bên ngoài lớp List nhưng không được khởi tạo, nên hàm dựng const\_iterator nhận tham số con trỏ đặt trong phạm vi protected.

Khai báo protected cho hàm dựng khiến lớp List không thể khởi tạo const\_iterator, giải pháp là đặt tính **friend** cho template List<Object>, cho phép List truy xuất tới thành viên non-public của const\_iterator, cũng như iterator.

protected:

/\*Dữ liệu thành viên khai báo protected để iterator có thể thừa kế và truy xuất,

\* nhưng không cho phép các lớp khác truy xuất tới.\*/

Node \*current; // con trỏ current trỏ tới node đại diện

Object & retrieve( ) const { return current->data; }

/\*Hàm dựng tham số con trỏ Node\*/

const\_iterator( Node \*p ) : current{ p } { }

friend class List<Object>;

};

1. class iterator

Triển khai thừa kế public ban cho **iterator** đầy đủ đặc tính (dữ liệu, phương thức thành viên) từ **const\_iterator**, và sự tương thích kiểu giúp iterator có thể dùng được bất kì nơi nào const\_iterator được triển khai.

template<typename Object>

class iterator : public const\_iterator{

public:

iterator( ){ }

/\* Không cần phải triển khai toán tử so sánh lại \*/

/\*Triển khai Accessor và Mutator cho toán tử \*,

bằng cách triệu hồi lại phương thức từ const\_iterator\*/

Object & operator\* ( ){ return const\_iterator::retrieve( ); }

const Object & operator\* ( ) const{ return const\_iterator::operator\*( ); }

//Quá tải tiền tố ++

iterator & operator++ ( ){

this->current = this->current->next;

return \*this;

}

//Quá tải hậu tố ++

iterator operator++ ( int ){

iterator old = \*this;

++( \*this );

return old;

}

protected:

/\*Triêu hồi hàm dựng của class thừa kế cho liên kết \*current \*/

iterator( Node \*p ) : const\_iterator{ p }{ }

friend class List< Object >;

};

Ngoài ra iterator được triển khai dữ liệu, phương thức mới và có thể ghi đè phương thức thừa kế.

Nói tới override, luôn có những tư tưởng không đúng thể hiện trong cách triển khai (phổ biến trong cách dùng từ khóa virtual trong code)

Chúng ta cần hạn chế mức tối thiểu cho các cú pháp thừa thãi vì không cần phải thêm dữ liệu nào mới, hay ý định thay đổi hành vi phương thức.

Trong khi triển khai thêm phương thức mới cho lớp iterator, dù hành vi tương đồng với phương thức trên const\_iterator, tránh dùng virtual.

Phương thức Accessor (quá tải toán tử \*) phải khai báo tường minh trong iterator để che giấu phương thức gốc trong const\_iterator.

1. class List

Trong triển khai Big-Five: hàm dựng không tham số và hàm dựng sao chép đều phải cấp phát cho header và tail node, nên cả 2 đều gọi phương thức init().

template<typename Object>

class List{

/\*Dữ liệu private gồm head/tail node, số phần tử theSize\*/

private:

int theSize;

Node \*head;

Node \*tail;

/\*init cấp phát head/tail node và cấu hình liên kết\*/

void init( ) {

theSize = 0;

head = new Node;

tail = new Node;

head->next = tail;

tail->prev = head;

}

public:

/\*TRIỂN KHAI BIG-5\*/

List( ) { init( ); }

~List( ) {/\*Hàm hủy gọi clear() xóa toàn bộ node, thu hồi head/tail node\*/

clear( );

delete head;

delete tail;

}

/\*copy constructor, gọi push\_back để nạp từng node\*/

List( const List & rhs ) {

init( );

for( auto & x : rhs )

push\_back( x );

}

/\*Move constructor\*/

List( List && rhs ) : theSize{ rhs.theSize }, head{ rhs.head }, tail{ rhs.tail } {

rhs.theSize = 0;

rhs.head = nullptr;

rhs.tail = nullptr;

}

/\*copy assignment, gọi Move constructor cho đối tượng copy và swap với \*this \*/

List & operator= ( const List & rhs ) {

List copy = rhs;

std::swap( \*this, copy );

return \*this;

}

/\*Move assignment\*/

List & operator= (List && rhs ) {

std::swap( theSize, rhs.theSize );

std::swap( head, rhs.head );

std::swap( tail, rhs.tail );

return \*this;

}

/\*Triển khai begin/end tương ứng cho iterator và const\_iterator

\* Cả 4 đều trả về hàm dựng protected của iterator và const\_iterator\*/

iterator begin( ){ return { head->next }; }

const\_iterator begin( ) const{ return { head->next }; }

iterator end( ) { return { tail }; }

const\_iterator end( ) const { return { tail }; }

/\*Phương thức lấy số phần tử, kiểm tra empty và xóa toàn bộ danh sách\*/

int size( ) const{ return theSize; }

bool empty( ) const { return size( ) == 0; }

void clear( ) {

while( !empty( ) )

pop\_front( );

}

/\*Accessor và Muatator cho phần tử đầu và cuối – 4 phương thức return \*iterator từ begin/end\*/

Object & front( ){ return \*begin( ); }

const Object & front( ) const{ return \*begin( ); }

Object & back( ){ return \*--end( ); }

const Object & back( ) const{ return \*--end( ); }

/\* PUSH\_ VÀ POP\_ DÙNG ITERATOR[BEGIN/END] ĐỂ ĐỊNH VỊ\*/

/\* Phương thức push\_ ủy nhiệm cho insert để xử lí node chèn\*/

void push\_front( const Object & x ){ insert( begin( ), x ); }

void push\_front( Object && x ){ insert( begin( ), std::move(x) ); }

void push\_back( const Object & x ){ insert( end( ), x ); }

void push\_back( Object && x ){ insert( end( ), std::move(x) ); }

/\* Phương thức pop\_ ủy nhiệm cho erase xử lí node xóa\*/

void pop\_front( ){ erase( begin( ) ); }

void pop\_back( ){ erase( --end( ) ); }

// Chèn x vào trước itr.

iterator insert( iterator itr, const Object & x ) {

Node \*p = itr.current;

theSize++;

return { p->prev = p->prev->next = new Node{ x, p->prev, p } };

}

// Chèn x vào trước itr.

iterator insert( iterator itr, Object && x ) {

Node \*p = itr.current;

theSize++;

return { p->prev = p->prev->next = new Node{ std::move( x ), p->prev, p } };

}

// Xóa item tại vị trí itr.

iterator erase( iterator itr ) {

Node \*p = itr.current;

iterator retVal{ p->next };

p->prev->next = p->next;

p->next->prev = p->prev;

delete p;

theSize--;

return retVal;

}

// Xóa item từ iterator from -> to.

iterator erase( iterator from, iterator to ) {

for( iterator itr = from; itr != to; )

itr = erase( itr );

return to;

}

};

Các phượng thức chèn, xóa sẽ trả về iterator đại diện cho node mới hay node tiếp theo node xóa.

Phiên bản erase thứ hai gọi lại phiên bản đầu tiên để xóa từng phần tử, trong vòng lặp không cần dùng toán tử ++ vì bản thân erase đã trả về iterator kế tiếp rồi.

1. Exception safe

Các lỗi phải được kiểm tra lại, ví dụ khi truyền iterator vào erase và insert, có 3 vấn đề:

* Iterator đã khởi tạo chưa?
* Iterator có đúng List không?
* Iterator có thể được ++ hay \* khi chúng đã ở vị trí kết thúc?

Iterator chưa khởi tạo dễ kiểm tra vì có \*current = nullptr, Iterator kết thúc thì luôn có \*next = nullptr nên không thể ++ hay \*.

Tuy nhiên, để kiểm tra iterator có thuộc List? Iterator nên triển khai thêm dữ liệu thành viên chứa con trỏ tới List.

public:

const\_iterator( ) : current{ nullptr }, theList { nullptr }{}

protected:

const List<Object> \*theList; // con trỏ tới List

Node \*current;

const\_iterator( const List<Object> & lst, Node \*p ): theList{ &lst }, current{ p }{}

/\*Error check: Iterator chưa khởi tạo\*/

void assertIsValid( ) const{

if( theList == nullptr || current == nullptr || current == theList->head )

throw IteratorOutOfBoundsException{ };

}

friend class List<Object>;

Chúng ta có thể thêm phương thức ném ra ngoại lệ nếu không thỏa điều kiện.

Tất cả lệnh gọi tới hàm dựng iterator và const\_iterator phản truyền thêm con trỏ \*this, ví dụ như begin:

const\_iterator begin( ) const {

const\_iterator itr{ \*this, head };

return ++itr;

}

Triển khai lại insert:

// Insert x before itr.

iterator insert( iterator itr, const Object & x ) {

itr.assertIsValid( );

if( itr.theList != this )

throw IteratorMismatchException{ };

Node \*p = itr.current;

theSize++;

return { \*this, p->prev = p->prev->next = new Node{ x, p->prev, p } };

}

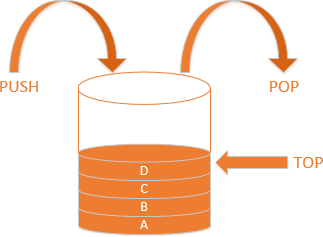
### Stack - Ngăn xếp

Stack – Ngăn xếp là cấu trúc dữ liệu quan trọng , là kiến thức không thể thiếu trong khoa học máy tính và được ứng dụng rất nhiều trong lập trình. Nó là kiểu dữ liệu cơ bản để giải những bài toán từ đơn giản đến phức tạp, nhiều bài toán phức tạp đã được đơn giản hóa đi rất nhiều nhờ loại cấu trúc dữ liệu này.

Vì trong bài này tôi xây dựng Stack bằng [Linked List](http://www.stdio.vn/articles/read/106-danh-sach-lien-ket-don)- các bạn có thể tham khảo thêm vấn đề - do đó bài viết này hướng đến những bạn có kiến thức về con trỏ và Linked List (danh sách liên kết).

##### Stack là gì?

Stack là một kiểu cấu trúc dữ liệu và cơ chế của nó là LIFO (Last In First Out) nghĩa là vào sau ra trước. Ta có thể hình dung Stack như một chồng đĩa ,ta chỉ có thể lấy chiếc đĩa ra hoặc thêm một chiếc đĩa khác vào trên đỉnh của nó và chiêc đĩa nằm trên đỉnh đó được gọi là Top.

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_4/104/ss_1.png)

##### Triển khai

Vì stack đơn thuần là List, nên list và vector đều hỗ trợ thao tác kiểu stack, 99% chúng là lựa chọn phù hợp. Stack được dùng cho các mục đích đặc biệt, các thao tác trên stack luôn là O(1),.

Có 2 cách triển khai phổ biến: dùng cấu trúc DSLK(list) hay dùng mảng (vector).

1. Dạng DSLK

Mỗi phần tử sẽ được chèn vào node đầu của danh sách. Và cũng bắt đầu xóa từ node đầu danh sách. Thao tác top() xác định node đầu của danh sách, trả về giá trị. Thỉnh thoảng pop() và top() được kết hợp là một.

Sometimes the pop and top operations are combined into one.

1. Dạng mảng

Dạng mảng được dùng phổ biến hơn. Nó dùng phương thức back(), push\_back(), pop\_back() từ vector, nên triển khai khá dễ dàng.

Chú ý rằng thơi gian thao tác trên mảng là một hằng số, nên sẽ nhanh hơn so với DSLK. Trên một vài máy, thao tác push và pop (của số nguyên) ,có thể viết bằng mã máy, triển khai trên thanh ghi nên còn nhanh hơn nhiều.

Thực tế là các máy tính hiện đại có tiến trình stack như một phần của bộ chỉ thị xử lí, khiến stack là cấu trúc dữ liệu phổ biến nhất trong khoa học máy tính, sau mảng.

##### Các phương thức của Stack

1. Cài đặt cấu trúc stack

Trong bài này tôi sử dụng DSLK để tạo cấu trúc stack.

1. struct Number{
2. int number;
3. Number \*pNextNum;
4. };
5. Number \*top;

Trên đây tôi tạo một struct tên là Number, trong struct này gồm có một field kiểu int để lưu trữ giá trị của một phần tử và con trỏ kiểu Number để trỏ tới phần tử kế tiếp trên nó trong stack. Con trỏ top kiểu Number dùng để trỏ tới phần tử trên cùng của stack.

1. isEmpty()

Phương thức isEmpty này giúp kiểm tra stack có rỗng hay không, nếu rỗng sẽ trả về true nếu không rỗng sẽ trả về false.

1. int isEmpty(){
2. if (top == NULL)
3. return true;
4. return false;
5. }
6. Push()

Phương thức Push là thêm một phần tử vào trên cùng của stack.

1. void pushNum(int value){
2. Number \*ptr = new Number;//tạo mới một phần tử
3. ptr->number = value; //gán giá trị cho phần tử
4. ptr->pNextNum = top; //phần tử này trỏ tới phần tử dưới nó trong stack
5. top = ptr; //đánh dấu phần tử này hiện đang nằm trên đỉnh của stack
6. }

Trước tiên ta tạo ra một con trỏ mới kiểu Number và gán giá trị cho phần tử này. Cho phần tử này liên kết đến phần tử nằm ngay dưới nó bằng cách sử dụng field pNextNum trỏ tới phần tử đó. Cuối cùng ta dùng con trỏ top trỏ tới phần tử vừa được thêm vào stack để có thể lấy phần tử đó ra.

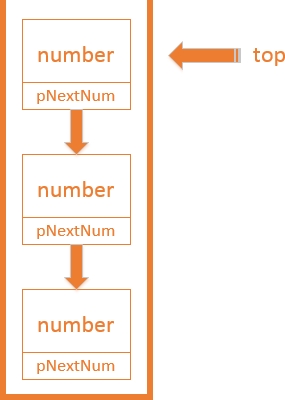
1. Pop()

Phương thức Pop là lấy một phần tử nằm trên đỉnh của stack.

1. int popNumber(){
2. int result = 0;
3. if (isEmpty()) {
4. return NULL; //nếu stack rỗng thì return về NULL
5. }
6. else {
7. Number \*ptr = top; //dùng một con trỏ trỏ tới phần tử đầu tiên của stack
8. result = top->number; //lấy giá trị
9. top = top->pNextNum; //gán con trỏ top cho phần tử ngay dưới nó
10. delete ptr; //delete phần tử vừa được lấy
11. return result; // trả kết quả
12. }
13. }

Nhiệm vụ của phương thức này là lấy một phần tử ở trên đỉnh của stack và trả về giá trị của phần tử đó. Ngay sau khi lấy một phần tử ra thì ta phải xóa phần tử đó đi để tránh memory leak.

Chúng ta có thể hình dung stack được xây dựng bằng linked list qua sơ đồ sau:

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_4/104/ss_2.png)

##### Ứng dụng stack

1. Kiểm tra kí hiệu

Trinh biên dịch kiểm tra cú pháp dòng lệnh, thường việc thiếu một kí hiệu nào đó khiến trình biên dịch dàn hàng trăm dòng ra phân tích mà không xác định đúng vị trí lỗi.

Sẽ cần một chương trình tốt để kiểm tra mọi thứ được cần bằng. Mọi cặp ngoặc tròn, vuông và phẩy phải tương ứng với nhau.

[()] là đúng, nhưng [(]) là sai. Chúng ta sẽ viết chương trình nhỏ kiểm tra, nó khá đơn giản, chỉ xét dấu ngoặc vuông, tròn và phẩy thôi, bỏ qua các kí tự khác.

Giải thuật là dùng stack:

* Tạo một stack
* Đọc kí tự cho tới khi hết file:
* Nếu có kí hiệu mở, push() vào stack.
* Nếu có kí hiệu đóng mà stack đang rỗng, báo lỗi.
* Ngược lại pop() stack. Nếu kí hiệu pop() không giống kí hiệu mở, báo lỗi.
* Khi tới hết file mà stack không rỗng, báo lỗi.

Ngoài ra bạn có thể phát triển thêm tính năng truy tìm lỗi bắt đầu từ đâu cho chương trình.

1. Biểu thức hậu tố toán tử

Hình dung bạn đang tính tiền các món hàng, thuế suất là 1.06, một số món không tính thuế, biểu thức như sau:

4.99 ∗ 1.06 + 5.99 + 6.99 ∗ 1.06 =

Cách tính thông thường sẽ là nhân 4.99 với 1.06, nhớ kết quả vào A1. Sau đó cộng với 5.99, lưu kết quả vào A1, nhân 6.99 với 1.06 và lưu kết quả vào A2, cuối cùng tính tổng A1 + A2, trả về kết quả vào A1.

Chúng ta có thể viết thành chuỗi thao tác như sau:

4.99 1.06 ∗ 5.99 + 6.99 1.06 ∗ +

Đây là cách viết kí hiệu toán tử hậu tố, hay kiểu tính Ba Lan ngược, và tìm ra trị số theo cách ta mô tả ở trên.

Giải thuật cho chương trình là dùng stack:

* khi con số được nhập, đẩy nó vào stack,
* khi toán tử được nhập, pop() hai con số ra, áp dụng toán tử, kết quả đẩy ngược lại vào stack.

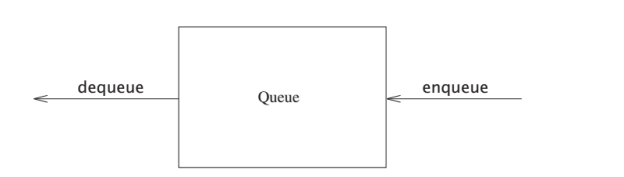
Thời gian tính của biểu thức hậu tố là O(N), vì các phần tử luôn lấy ra từ top và đẩy ngược vào top. Giải thuật khá đơn giản, không cần phải tuân theo luật ưu tiên nào, đây là ưu điểm.

### Queue – Hàng đợi

##### Queue là gì

Giống như stack, queue vẫn là kiểu danh sách. Tuy nhiên, queue hoạt động theo nguyên tắc FILO (First in last out)

Thao tác cơ bản trên queue là xếp hàng (enqueue), chèn item vào cuối danh sách. Dequeue: lấy item ra ở vị trí đầu sanh sách.



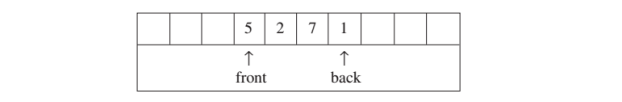
##### Triển khai

Giống như stack, queue có thể triển khai trên list hay vector đều được. DSLK và mảng đều có hiệu suất O(1) trên 2 thao tác.

DSLK dễ dàng hơn. Chúng ta sẽ thảo luận về mảng trên queue.

Tổ chức dữ liệu queue:

* chúng ta duy trì hai vị trí là front và back, đại diện cho 2 điểm cuối của queue.
* Dữ liệu về số phần tử: currentSize.



Thao tác:

* Enqueue phần tử x: tăng currentSize, giảm back, cho theArray[back] = x
* Dequeue: trả về theArray[front], giảm currentSize, sau đó tăng front

Check lỗi:

Sau 10 enqueue, hàng đợi sẽ đầy, back sẽ ở vị trí cuối, thao tác enqueue kế tiếp sẽ không tồn tại vị trí. Tuy nhiên, vẫn xảy ra trường hợp chỉ có vài phần tử trong queue, khi bị dequeue.

Giải pháp đơn giản là bất cứ khi nào front hay back đi tới cuối mảng, nó được wrap về vị trí đầu tiên.

Đây gọi là triển khai mảng vòng.

|  |  |
| --- | --- |
| Phần code triển khai wrap về vị trí đầu là ít dù nó có thể chạy gấp đôi thời gian.  Nếu tăng back hay front, lọt qua mảng, sẽ được reset về vị trí đầu mảng.  Một số lập trình viên dùng cách khác để minh họa front và back.  Ví dụ: một vài người không duy trì số phần tử, queue trống khi back = front – 1. Kích thước được tính dựa trên việc so sánh back và front.  Cách trên là một mẹo khá hay, nhưng trong một số trường hợp đặc biệt, phải cận thận:  Nếu không duy trì currentSize như một thành viên dữ liệu, queue đầy khi theArray.capacity() – 1 = 0  Trên ứng dụng, nếu bạn luôn đảm bảo số enqueue không lớn hơn capacity của queue, việc wrap là không cần thiết.  GIống như stack, dequeue chỉ được thực thi khi queue không trống. |  |

##### Ứng dụng

Khá nhiều giải thuật cần dùng hàng đợi để đạt hiệu suất chạy cao nhất. Một vài trong đó là lý thuyết đồ thị.

Một vài ví dụ cụ thể:

* Khi jobs đưa vào printer, chúng sắp theo hàng đợi
* Trong cuộc sống: sắp hàng lấy vé xem phim, các cuộc gọi tới tổng đài, nhận phiếu ghi số thứ tự trong ngân hàng, bệnh viện…

Có cả một phân nhánh toán học nghiên cứu lý thuyết hàng đợi trong tính toán xác suất: bao lâu người dùng phải đợi line điện thoại, bao lâu thì line có để phục vụ, …

Câu trả lời lệ thuộc vào số người dùng gọi tới line này và thời gian phục vụ 1 người dùng trung bình. Tất cả đều là tham số của hàm phân bổ xác suất.

Theo phân tích, trường hợp này đơn giản như sau:

* 1 line sẽ có 1 nhân viên trực
* Nếu nhân viên bận, người gọi sẽ được cho vào hàng đợi line thoại (có quy định số lượng tối đa).

Vấn đề trên khá quan trọng trong doanh nghiệp, người gọi dễ bực khi tắc line thoại. Làm sao để xác định số lượng tổng đài viên vừa đủ để phục vụ số người gọi?

Chúng ta phải dùng queue để mô phỏng, với nhiều giá trị nhập vào. Giả sử k là số tổng đài viên, chọn k tối đa hay tối thiểu, để cho ra thời gian chờ tối ưu.

Khi số người dùng tăng lên, chúng ta mới thấy tầm quan trọng của cấu trúc đơn giản như stack và queue.

### Cấu trúc cây

Đối với lượng dữ liệu lớn, thời gian truy xuất tuyến tính trên DSLK là khó chấp nhận. Chương này, giới thiệu về cấu trúc cây, có thời gian truy xuất trung bình là O(logN).

Cấu trúc dữ liệu cây nhị phân tìm kiếm là nền tảng cơ bản để triển khai hai bộ thư viện : “set” và “map”, được dùng trong nhiều ứng dụng.

##### Giới thiệu

Cây có thể định nghĩa nhiều cách, nhưng thường là dùng đệ qui.

Kiến trúc cây sẽ gồm: node gốc root, và các cây con. Đường liên kết giữa Node cha và node con gọi là các cạnh

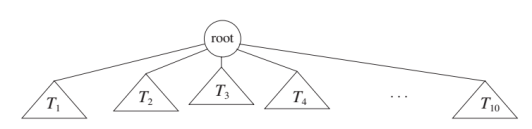
 Giả sử cây có N node, một node root, như vậy có N-1 cạnh. Vì ngoại trừ node root, các node khác đều chỉ có một node cha.

Figure : cây đệ qui

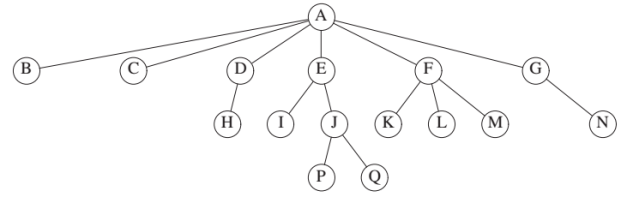


Figure : cây tổng quát

* Các node nằm dưới cùng không có node con gọi là node lá: H, I, P, Q, K ,L , M, N.
* Các node có cùng node cha gọi là liền kề: B, C, D, E, F, G (node root A) hay I, J (node cha E).
* Đường đi từ node n1 tới nk đi qua : n1, n2 ….,nk. Độ dài của đường đi là số cạnh trên đường (k-1).
* Cây luôn chỉ có một đường duy nhất từ node root tới mỗi node.
* Độ sâu của node ni là độ dài đường đi từ root tới ni.
* Độ cao của node ni là độ dài đường đi dài nhất từ ni tới node lá: E có độ sâu là 1, độ cao là 2.
* Độ sâu của cây bằng độ sâu của node lá sâu nhất, nó cũng là chiều cao của cây.

Nếu có đường đi từ n1 tới n2, thì n1 là cha của n2, n2 là con của n1.

Nếu n1 khác n2, thì n1 chắc chắn là cha của n2, n2 chắc chắn là con của n1.

##### Ứng dụng cây trong hệ thống File

1. Cấu trúc cây

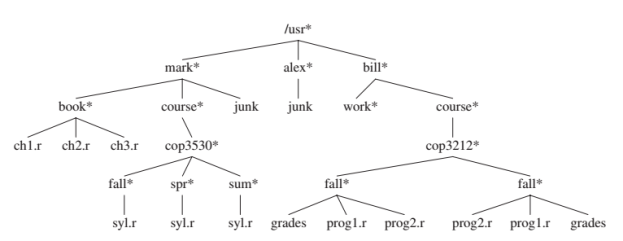


Figure : hệ thống phân cấp thư mục Unix

Một trong những ứng dụng phổ biến là hệ thống thư mục phân cấp của hệ điều hành, gồm Unix và Dos.

Với Unix, root của thư mục là /usr (dấu hoa thị \* ám chỉ rằng đây là thư mục).

File /usr/mark/book/ch1.r : dấu (/) trước mỗi node ám chỉ một cạnh, kết quả là đường dẫn đầy đủ.

Hai file khác thư mục có thể trùng tên. Thư mục trong hệ thống file Unix đơn thuần là file với danh sách các node con, nên thư mục vẫn định nghĩa là một node.

Mỗi thư mục trên Unix vẫn có một entry trỏ tới chính nó và entry khác trỏ tới node cha. Do đó hệ thống file Unix là một cấu trúc giống như cây mà thôi.

1. Thuật toán

Phương thức đệ qui listAll() xuất ra tên tất cả file trong thư mục đó.(cùng với thông tin không phải dạng ASCII)

void FileSystem::listAll( int depth = 0 ) const{

printName( depth ); // In ra tên đối tượng

// Xét đối tượng có phải thư mục

if( isDirectory( ) ) {

//đệ qui hàm cho mỗi file c có trong thư mục

c.listAll( depth + 1 );

}

}

1. Duyệt từ trên xuống (preoder traverse)

Ví dụ thư mục có độ sâu *di* thì tên nó sẽ có *di* tabs. listAll() sẽ bắt đầu duyệt từ root (depth = 0), đây là giá trị tham số mặc định.

Thuật toán đệ qui khá đơn giản:

* Tên file in ra cùng với số tabs.
* Nếu đối tượng là thư mục, đệ qui hàm cho tất cả node con, từng cái một.

Cách thức duyệt ngang như trên được biết như là duyệt theo thứ tự trước :

* In ra tên node cha trước rồi mới in tên mỗi node con trong thư mục với hàm printName()
* Dòng lệnh đệ qui c.listAll() sẽ duyệt tiếp các node con có độ sâu lớn hơn.

Như vậy tổng cộng mỗi node trong cây chỉ duyệt qua một lần, thời gian chạy sẽ là O(n).

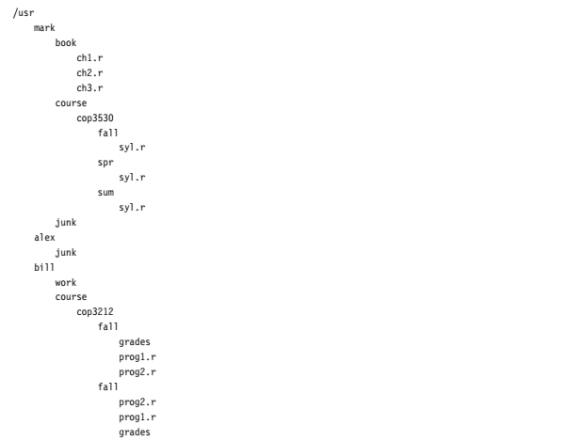


Figure 4: liệt kê thư mục

1. Duyệt từ dưới lên (postorder traverse)

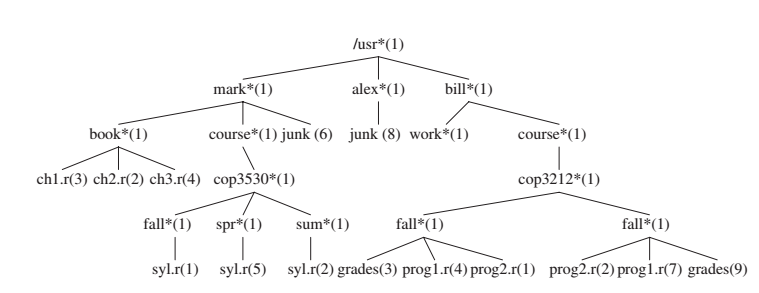


Figure 5: Thư mục Unix với file size khi duyệt từ dưới

Hình bên trên minh họa kích thước mỗi node con được trả về bởi giải thuật đệ qui

int FileSystem::size( ) const{

int totalSize = sizeOfThisFile( );

if( isDirectory( ) ) {

//đệ qui cho mỗi file c trong thư mục

totalSize += c.size( );

}

return totalSize;

}

Phương thức thứ 2 là duyệt từ dưới lên:

* Trường hợp không phải thư mục, return về totalSize
* Trường hợp là thư mục, đệ qui mỗ node con c.size(), đồng thời return tổng kích thước node cha với các node con.

##### Cây nhị phân

1. Giới thiệu

Cây nhị phân là một kiểu cấu trúc dữ liệu được định nghĩa trên nguyên lý đệ qui và con trỏ. Vì độ sâu của cây tối đa là O(logn), nên không cần bận tâm về tràn bộ đệm của stack khi đệ qui.

Để triển khai cây nhị phân, bạn cần kiến thức về đệ qui và danh sách liên kết.

DSLK (danh sách liên kết) mất khá nhiều thơi gian để tìm một phần tử trong danh sách.

Việc đầu tiên là phải sắp xếp danh sách: giúp bạn tìm kiếm nhanh hơn. Nhưng vẫn chậm khi chèn thêm dữ liệu và luôn phải sắp xếp lại.

Tốc độ tìm kiếm luôn là chỉ tiêu quan trọng. Ví dụ

1. MMORPG World of Warcraft: Bạn phải tối ưu giải thuật tìm kiếm giúp người chơi đăng nhập thật nhanh vào game
2. Khi xử lí thẻ tín dụng, có hàng triệu giao dịch mỗi giờ- bạn phải tìm tài khoản đúng số thẻ tín dụng tức thì.
3. Nếu bạn làm việc với smartphone – thao tác phổ biến là tra cứu danh bạ.

Ý tưởng cơ bản là lưu dữ liệu theo kiểu DSLK – nhưng với cấu trúc vùng tinh vi, dễ dàng tìm kiếm hơn.

1. Cấu trúc cây nhị phân

Bạn có thể có hơn một node cùng lúc, tức có hai “next” node, một đại diện cho phần tử nhở hơn node hiện tại, một lớn hơn node hiện tại. Ta gọi đây là cấu trúc cây nhị phân.

Mỗi next node gọi là node con so với node cha là node hiện tại, bạn hình dung cây nhị phân như sau:



Các node con trái luôn nhỏ hơn node cha, node con phải thì luôn lớn hơn. Node 10 là node gốc của cây. Mỗi node có thể có cấu trúc cây riêng của nó, gọi là cây con.

LUẬT THIẾT KẾ CÂY NHỊ PHÂN

Lưu ý:

* Cây nhị phân có thuộc tính là mỗi node con cũng chính là một cây nhị phân
* Node con trái nhỏ hơn node hiện tại, node con phải lớn hơn node hiện tại

GIẢI THUẬT TÌM KIẾM

Hai điều trên là cơ sở để thiết kế giải thuật tìm kiếm một node trong cây:

* Đầu tiên, xét giá trị node hiện tại, nếu bằng mục tiêu tìm kiếm thì xong
* Nếu mục tiêu nhỏ hơn node hiện tại, duyệt sang trái, ngược lại lớn hơn thì duyệt qua phải

Ý tưởng nữa là phải cân bằng cây, tổng node trái cũng phải bằng tổng node phải. Tức là mỗi nhánh cây nhị phân con sẽ có kích thước bằng một nửa kích thước toàn cây hiện tại, theo đó, giải thuật tra cứu đã loại bỏ 50% khả năng không phù hợp (một nửa cây).

Với cây 1000 node, qua một bước duyệt, bạn sẽ loại bỏ 500 node, tiếp tục là 250 ,..

Như vậy số lần duyệt là: y = log2 𝑛. ( n là số phần tử)

Số lần duyệt cực kì nhỏ (4 tỉ node chỉ mất 32 lần duyệt), nhanh hơn hàng trăm lần so với DSLK.

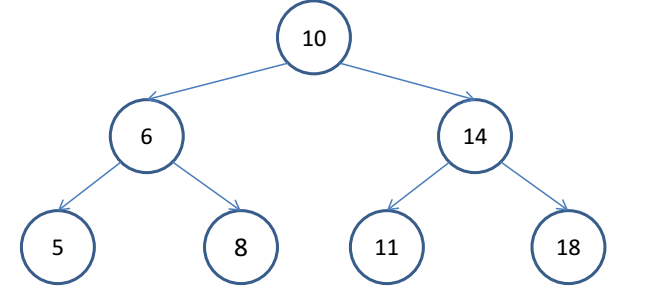
Tuy nhiên, nếu cây không cân bằng, bạn không thể nào cắt cây chính xác một nửa được. Tình huống xấu nhất là mọi node chỉ có đúng một node con, hay chỉ là DSLK với nhiều con trỏ.

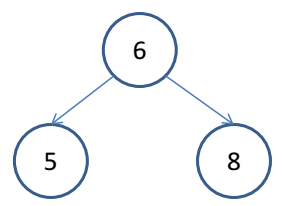
1. Quy tắc chung

Chúng ta sẽ bàn về quy tắc chung khi triển khai cây nhị phân:

* Một cây rỗng, đại diện bằng NULL. Chúng ta sẽ không vẽ liên kết tới cây NULL
* Khi muốn nói về cây con nào đó, ta dùng: < cây tẻ nhánh bởi [giá trị node cha] >.

Ví dụ



<cây tẻ nhánh bởi 6> sẽ là:

1. Triển khai

Trước tiên là khai báo cấu trúc node:

struct node{

int key\_value;

node \*p\_left;

node \*p\_right;

};

Node chứa giá trị tương trưng đơn giản integer, và hai node con trái, phải.

Đây là một số hàm thao tác cơ bản trên cây nhị phân

node\* insert (node\* p\_tree, int key);//chèn node có giá trị key

node \*search (node\* p\_tree, int key);//tìm node có giá trị key

void destroyTree (node\* p\_tree);//xóa cây con

node \*remove (node\* p\_tree, int key);// hủy toàn bộ cây nhị phân

1. Chèn cây

Chúng ta sẽ bắt đầu chèn dùng giải thuật đệ qui. Vì mỗi cây chứa hai cây con, nên bản thân cấu trúc cây đã đệ qui theo lẽ tự nhiên.

Giải thuật là:

* Nếu cây rỗng, tạo mới, trả về node mới tạo
* Nếu giá trị lớn hơn node hiện tại, chèn nó vào cây con phải, trả về node hiện tại
* Ngược lại chèn vào cây con trái node hiện tại, trả về node hiện tại.

Tham số gồm key và cây (có thể rỗng), hàm trả về cây chứa giá trị chèn vào:

node\* insert (node \*p\_tree, int key){

// Trường hợp suy biến: đã duyệt tới cây rỗng, ta chèn node vào đây và trả về node

if ( p\_tree == NULL ) {

node\* p\_new\_tree = new node;

p\_new\_tree->p\_left = NULL;

p\_new\_tree->p\_right = NULL;

p\_new\_tree->key\_value = key;

return p\_new\_tree;

}

// Xác định rẽ nhánh trái hay phải

if( key < p\_tree->key\_value ) {

// Nếu rẽ nhánh trái, ta gọi hàm đệ qui cho node con trái của node hiện tại

p\_tree->p\_left = insert( p\_tree->p\_left, key );

}

else {

// Nếu rẽ nhánh phải, gọi hàm đệ qui cho node con phải của node hiện tại

p\_tree->p\_right = insert( p\_tree->p\_right, key );

}

// Cuối cùng trả về node hiện tại cho trường hợp chèn vào cây con trái hay phải.

return p\_tree;

}

1. Tìm cây

Giải thuật cũng giống như khi chèn cây:

* Nếu node hiện tại là mục tiêu (nếu tìm thấy) hay cây rỗng (không tìm thấy), trả về node hiện tại
* Nếu node hiện tại không phải node cần tìm, lựa chọn để duyệt cây con trái hay phải

Triển khai:

node \*search (node \*p\_tree, int key){

// Nếu cây rỗng, tức không tìm thấy!

if ( p\_tree == NULL ) {

return NULL;

}

// Trường hợp tìm thấy!

else if ( key == p\_tree->key\_value ) {

return p\_tree;

}

// Ngược lại, lựa chọn duyệt cây con trái hay phải

else if ( key < p\_tree->key\_value ) {

return search( p\_tree->p\_left, key );

}

else{

return search( p\_tree->p\_right, key );

}

}

1. Hủy toàn bộ cây

Giải thuật sẽ hủy hai cây con của node hiện tại và sau đó xóa luôn node hiện tại.

void destroy\_tree (node \*p\_tree){

if ( p\_tree != NULL ){

destroy\_tree( p\_tree->p\_left );

destroy\_tree( p\_tree->p\_right );

delete p\_tree;

}

}

1. Xóa một node

Giải thuật xóa một node trong cây nhị phân khá phức tạp, nhưng tương tự như các triển khai trước:

* Nếu đi tới cây rỗng, tức không tìm thấy, chúng ta trả về.
* Nếu tìm thấy, hủy nó.
* Nếu giá trị cần nằm cây con trái, đệ qui hàm cho cây con trái
* Ngược lại , đệ qui hàm cho cây con phải

1. Sơ đồ triển khai:

node \* remove (node\* p\_tree, int key){

if ( p\_tree == NULL ){

return NULL;

}

if ( p\_tree->key\_value == key ){

// what to do?

}

else if ( key < p\_tree->key\_value ){

p\_tree->p\_left = remove( p\_tree->p\_left, key );

}

else{

p\_tree->p\_right = remove( p\_tree->p\_right, key );

}

return p\_tree;

}

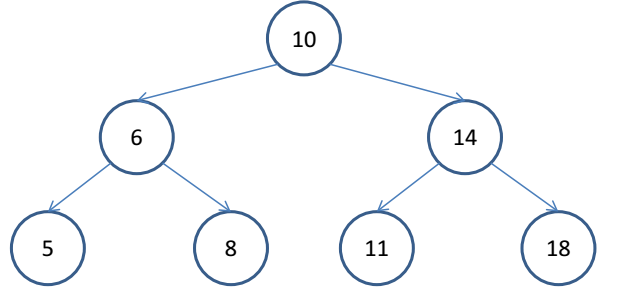
Rắc rối bắt đầu từ trường hợp suy biến: tìm thấy node.

Trước khi hủy node, nhớ rằng cây nhị phân vẫn phải được duy trì theo qui tắc: mọi node trong cây con trái luôn nhỏ hơn node hiện tại và mọi node trong cây con phải.

Cơ bản, chúng ta phải xử lí 3 trường hợp như sau:

1. Node hủy không có cây con nào: khá đơn giản, chỉ cần hủy node và trả về NULL
2. Node hủy có một cây con : chỉ cần hủy node và return lại cây con đó
3. Node hủy có hai cây con:

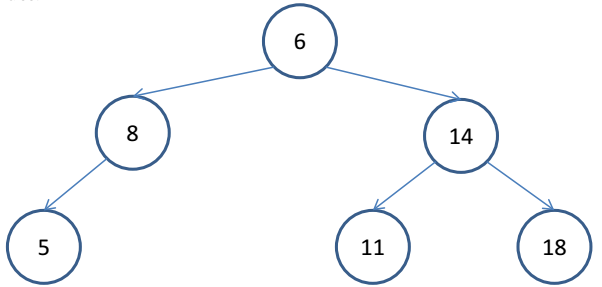
Đối với trường hợp thứ 3. Xét ví dụ cần xóa node 10 trong cây



Ta có hai cách làm

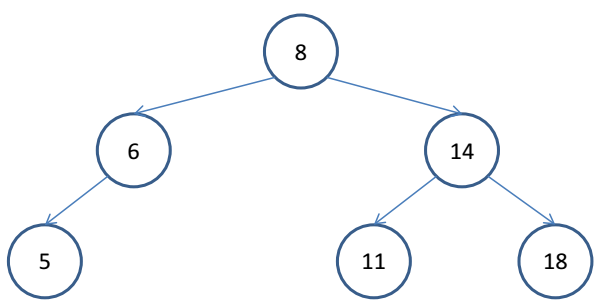
1. Đưa cây con trái lên thay

Nếu bạn đưa node 6 lên thay:



Chúng ta sẽ không bao giờ tìm được node 8, vì node 8 lớn hơn node 6 và lại nằm ở cây con trái.

Để duy trì cây nhị phân, chúng ta phải đưa node 8 lên



1. tìm node lớn nhất cây con trái

Giải thuật là tìm node có giá trị lớn nhất của cây con trái đưa lên thay node bị hủy, do đó chúng ta sẽ triển khai hàm find\_max(), dựa trên quy tắc cây con phải luôn lớn hơn, hàm sẽ duyệt các cây con phải tới khi gặp NULL, trả về cây con giá trị lớn nhất.

* Nếu cây con hiện tại là NULL, tức không có cây con phải, return NULL
* Nếu cây con phải của node hiện tại là NULL, trả về node hiện tại
* Ngược lại đệ qui hàm cho cây con phải

Triển khai:

node\* find\_max(node\* p\_tree) {

if (p\_tree == NULL) {

return NULL;

}

if (p\_tree->p\_right == NULL) {

return p\_tree;

}

return find\_max(p\_tree->p\_right);

}

1. xóa node lớn nhất cây con trái

Chúng ta lại cần huỷ liên kết max\_node trong cây con, tức chỉ cần node cha của max\_node trỏ tới cây con trái của nó là xong.

Hàm remove\_max\_node() nhận tham số vào: cây con trái node cần xóa và con trỏ tới max\_node. Hàm có nhiệm vụ duyệt cây con tới khi tìm thấy max\_node, sau đó nối liên kết cây con max\_node vào node cha của nó, tức return cây con phải của max\_node về.

**Lưu ý: chúng ta không hủy max\_node.**

Giải thuật:

* Trường hợp suy biến: p\_tree = max\_node, trả về cây con trái max\_node là xong.
* Trường hợp suy biến: p\_tree = NULL, tức cây con trái node cần xóa không có cây con phải, return NULL.
* Ngược lại duyệt tiếp cây con phải cho p\_tree->right.

node\* remove\_max\_node(node\* p\_tree, node\* p\_max\_node) {

// không tồn tại cây con phải

if (p\_tree == NULL) {

return NULL;

}

// tìm thấy max\_node

if (p\_tree == p\_max\_node) {

// Cho cả trường hợp cây con trái của max\_node tồn tại hay không, chúng ta chỉ cần return

// cây con đó về

return p\_max\_node->p\_left;

}

// Trường hợp chưa tìm thấy, đệ qui hàm tiếp cho cây con phải

p\_tree->p\_right = remove\_max\_node(p\_tree->p\_right, p\_max\_node);

return p\_tree;

}

1. lắp ráp

Với sự trợ lực hàm trên, chúng ta dễ dàng hoàn thành nốt hàm remove:

node\* remove(node\* p\_tree, int key) {

if (p\_tree == NULL) {

return NULL;

}

if (p\_tree->key\_value == key) {

// Trường hợp node cần xóa có 1 cây con hay không có cây con nào

if (p\_tree->p\_left == NULL) {

node\* p\_right\_subtree = p\_tree->p\_right;

delete p\_tree;

// Trả về cây con phải node cần xóa

return p\_right\_subtree;

}

if (p\_tree->p\_right == NULL) {

node\* p\_left\_subtree = p\_tree->p\_left;

delete p\_tree;

// cây con trái sẽ luôn tồn tại, vì không thỏa điều kiện if trên

return p\_left\_subtree;

}

// Trường hợp 3, node xóa có hai cây con

node\* p\_max\_node = find\_max(p\_tree->p\_left);

// Xóa liên kết max\_node khỏi cây con

// Đồng thời tạo liên kết mới cho max\_node

p\_max\_node->p\_left = remove\_max\_node(p\_tree->p\_left, p\_max\_node);

p\_max\_node->p\_right = p\_tree->p\_right;

delete p\_tree;

return p\_max\_node;

} else if (key < p\_tree->key\_value) {

p\_tree->p\_left = remove(p\_tree->p\_left, key);

} else {

p\_tree->p\_right = remove(p\_tree->p\_right, key);

}

return p\_tree;

}

**Lưu ý:**  Trong cả ba trường hợp (tìm thấy, không tồn tại, chưa tìm thấy node xóa), chúng ta đều phải trả về con trỏ cây (hay NULL).

Vì trong quá trình đệ qui, chúng ta cũng vừa cấu trúc lại cây nhị phân từ node bắt đầu tìm tới node thay thế.

1. Đưa cây con phải lên thay

Cũng tương tự như cách một. Triển khai:

//ham tim min\_node

node\* find\_min(node\* p\_tree) {

//Truong hop nay khong the xay ra

if (p\_tree == NULL) return NULL;

//Truong hop tim thay

if (p\_tree->p\_left == NULL) {

return p\_tree;

}

//Truong hop chua tim thay, de qui ham

return find\_min(p\_tree->p\_left);

}

//ham xoa min\_node khoi cay con

node\* remove\_min\_node(node\* p\_tree, node\* p\_min\_node) {

//Truong hop nay khong the xay ra

if (p\_tree == NULL) return NULL;

//Truong hop tim thay

if (p\_tree == p\_min\_node) {

//tra ve cay con phải min\_node

return p\_min\_node->p\_right;

}

//Truong hop chua tim thay, de qui ham

p\_tree->p\_left = remove\_min\_node(p\_tree->p\_left, p\_min\_node);

return p\_tree;

}

//ham huy node

node\* remove2(node\* p\_tree, int key) {

//Truong hop node xoa khong ton tai

if (p\_tree == NULL) return NULL;

//Truong hop tim thay

if (p\_tree->key\_value == key) {

//Truong hop node xoa co 1 cay con hay khong co cay con nao

if (p\_tree->p\_left == NULL) {

node\* p\_right\_subtree = p\_tree->p\_right;

delete p\_tree;

return p\_right\_subtree;

}

if (p\_tree->p\_right == NULL) {

node\* p\_left\_subtree = p\_tree->p\_left;

delete p\_tree;

return p\_left\_subtree;

}

//Truong hop node xoa co ca 2 cay con

//tim min\_node nam trong cay con trai node xoa

node\* p\_min\_node = find\_min(p\_tree->p\_right);

//Tai cau truc cay con phai node xoa, dua p\_min\_node len thay node xoa

p\_min\_node->p\_right = remove\_min\_node(p\_tree->p\_right, p\_min\_node);

p\_min\_node->p\_left = p\_tree->p\_left;

delete p\_tree;

return p\_min\_node;

}

//Truong hop tim chua thay, de qui ham

if (key < p\_tree->key\_value) {

p\_tree->p\_left = remove2(p\_tree->p\_left, key);

} else {

p\_tree->p\_right = remove2(p\_tree->p\_right, key);

}

//tra ve p\_tree

return p\_tree;

}

1. ứng dụng của binary trees

Nói chung, sẽ có hai mục đích chính khi tìm kiếm:

* Đầu tiên là kiểm tra xem đã tồn tại giá trị chưa: ví dụ khi người dùng đăng kí tài khoản mới, bạn phải kiểm tra tài khoản tồn tại hay chưa. Và user\_name thực ra là kiểu chuỗi, sẽ mất thời gian hơn rất nhiều so với kiểu số, nhất là duyệt trên hàng triệu tài khoản khác nhau. Thêm nữa, việc kiểm tra nhanh đem lại sự thoải mái cho người dùng.
* Thứ hai là khi bạn có một số dữ liệu cần lưu trữ tương ứng với các giá trị : kiểu dữ liệu này gọi là cấu trúc **Map**. Một Map lưu một khóa và một giá trị tương ứng với khóa đó ( giá trị này có thể là một kiểu dữ liệu hay một cấu trúc phức tạp chứa nhiều thông tin).  
  Ví dụ game World of Warcraft(WOW) hay bất kì game online multiplayer nào cũng sẽ cần một Map từ user\_name tới pass\_word (\*), để quản lý đăng nhập và tính điểm người chơi online. Mỗi lần bạn đăng nhập, WOW tìm user\_name của bạn trong Map và tìm pass\_word tương ứng, so sánh pass\_word này với chuỗi bạn nhập vào, nếu khớp, lấy tất cả thông tin nhân vật và để bạn chơi game.

(\*): Trên thực tế, bản thân pass\_word nằm trong Map không phải mật khẩu lúc bạn đăng kí, mà chỉ là phiên bản mật khẩu đã được băm ra. Giải thuật băm trả về chuỗi kí tự không thể phục hồi từ chuỗi gốc. Dù có biết chuỗi kí tự băm, cũng không thể suy ra chuỗi gốc, tránh nguy cơ bị lộ mật khẩu. Khi bạn đăng nhập pass\_word, máy tính sẽ băm chuỗi bạn nhập và so sánh với phiển bản trong Map.

Bạn có thể triển khai cấu trúc Map như vậy bằng cây nhị phân:

* Yêu cầu cây nhị phân phải dùng khóa (tên người dùng) cho việc chèn nodes và lưu trữ giá trị thông tin tương ứng (mật khẩu) trong cùng node đó.

Ứng dụng lý thuyết về Map có mặc ở mọi nơi. Ví dụ: công ty thẻ tin dụng muốn dùng Map cho vài kiểu sắp xếp – tức cứ mỗi khi bạn mua hàng qua thẻ tín dụng, dữ liệu về tài khoản của bạn cần thay đổi.

Có hàng chục triệu người dùng thẻ tín dụng, việc quét hàng loạt mã số thẻ trên mọi giao dịch sẽ khiến hoạt động giao dịch ngưng trệ trên phạm vi toàn cầu.

Ý tưởng cơ bản là bạn cần có thể tìm tài khoản cân bằng thật nhanh, qua mã số thẻ. Để làm được, phải dùng cây nhị phân xây dựng Map cho mỗi mã số thẻ tương ứng với tài khoản cân bằng .

Bây giờ mỗi giao dịch xảy ra, máy tính sẽ thực hiện duyệt cây nhị phân cho mã số thẻ sau đó cập nhật thông tin.

1 triệu tài khoản sẽ trải qua log2(1 triệu) tương đương 20 lần duyệt. Nhanh hơn 50 ngàn lần so với duyệt tuyến tính trên danh sách liên kết.

Một lưu ý là thông tin chi tiết tài khoản luôn nằm trong cơ sở dữ liệu, chứ không phải vùng nhớ. Cấu trúc Map có thể sẽ phức tạp hơn tùy theo thiết kế từng công ty. Điều quan trọng là ý tưởng cây nhị phân và Map là nền tảng để xây dựng cấu trúc tinh vi.

Cuối cùng là ví dụ trên điện thoại đi động: tra cứu danh bạ theo số điện thoại. Sổ danh bạ có thể hưởng lợi từ cấu trúc như cây nhị phân và Map thường được xây dựng trên một cấu trúc cây nhị phân cho phép tra cứu nhanh

Ngoài ra, cấu trúc bảng băm cũng được dùng để triển khai Map.

##### Cây nhị phân – template class

1. Interface

Bên dưới là interface của lớp template BinarySearchTree.

template <typename Comparable>

class BinarySearchTree{

public:

/\*1 số Thao tác public trên cây sẽ gọi các hàm đệ qui private\*/

BinarySearchTree( );

BinarySearchTree( const BinarySearchTree & rhs );

BinarySearchTree( BinarySearchTree && rhs );

~BinarySearchTree( );

//

const Comparable & findMin( ) const;

const Comparable & findMax( ) const;

bool contains( const Comparable & x ) const;

bool isEmpty( ) const;

void printTree( ostream & out = cout ) const;

void makeEmpty( );

void insert( const Comparable & x );

void insert( Comparable && x );

void remove( const Comparable & x );

//

BinarySearchTree & operator=( const BinarySearchTree & rhs );

BinarySearchTree & operator=( BinarySearchTree && rhs );

private:

struct BinaryNode {

Comparable element;

BinaryNode \*left;

BinaryNode \*right;

BinaryNode( const Comparable & theElement, BinaryNode \*lt, BinaryNode \*rt )

: element{ theElement }, left{ lt }, right{ rt } { }

BinaryNode( Comparable && theElement, BinaryNode \*lt, BinaryNode \*rt )

: element{ std::move( theElement ) }, left{ lt }, right{ rt } { }

};

/\* Dữ liệu thành viên chứa con trỏ \*root; = nullptr tức là cây rỗng \*/

BinaryNode \*root;

//

void insert( const Comparable & x, BinaryNode \* & t );

void insert( Comparable && x, BinaryNode \* & t );

void remove( const Comparable & x, BinaryNode \* & t );

//

BinaryNode \* findMin( BinaryNode \*t ) const;

BinaryNode \* findMax( BinaryNode \*t ) const;

//

bool contains( const Comparable & x, BinaryNode \*t ) const;

void makeEmpty( BinaryNode \* & t );

void printTree( BinaryNode \*t, ostream & out ) const;

//

BinaryNode \* clone( BinaryNode \*t ) const;

};

Việc tìm kiếm sẽ dựa trên toán tử <, nên phải có quá tải so sánh cho tham số template Comparable.

Trường hợp đặc biệt, item x khớp y nếu xảy ra cả 2 trường hợp x < y và y<x false.

Tham số template sẽ là đối tượng có cấu trúc dữ liệu phức tạp, có thể so sánh được dựa trên dữ liệu thành viên.

Chúng ta có 2 kỹ thuật triển khai toán tử so sánh cho lớp: quá tải toán tử hoặc dùng đối tượng hàm.

1. Public method

Hình dung các phương thức public gọi các private đệ qui:

/\* trả về true nếu tìm thấy x trên cây\*/

bool contains( const Comparable & x ) const{return contains( x, root );}

/\* chèn x, nếu giá trị trùng thì bỏ qua\*/

void insert( const Comparable & x ){insert( x, root );}

/\* Xóa x khỏi cây, nếu không có thì chẳng làm gì cả\*/

void remove( const Comparable & x ){remove( x, root );}

Một vài phương thức private dùng kỹ thuật truyền con trỏ sử dụng cách gọi tham chiếu. Cho phép hàm thành viên public truyền con trỏ tới node root tới hàm thành viên private.

Hàm đệ qui có thể thay đổi giá trị của root để root trỏ tới node khác.

Chúng ta sẽ bàn thêm kỹ thuật này trong phần insert.

1. Contain

Khi cây rỗng hoặc không tìm thấy, contains trả về false.

/\*Private contains kiểm tra x có trên cây con t không\*/

bool contains( const Comparator & x, BinaryNode \*t ) const{

if( t == nullptr )

return false; // Không thấy trả về false

else if( x < t->element )

return contains( x, t->left );

else if( t->element < x )

return contains( x, t->right );

else

return true; // Tìm thấy trả về true

}

Điểm quan trọng là phải đặt kiểm tra cây rỗng lên trên cùng – để tránh lỗi runtime truy xuất thành viên nullptr của cây rỗng.

Triển khai bên trên sử dụng quá tải so sánh bé hơn (<) của đối tượng x. Bên dưới là phiên bản dùng functor:

**Triển khai dùng functor Comparator(const Object&, const Object&) so sánh 2 đối tượng Object**

template <typename Object, typename Comparator=less<Object>>

class BinarySearchTree {

public:

private:

BinaryNode \*root;

Comparator isLessThan; //trả về true nếu tham số 1 < tham số 2. False nếu ngược lại.

bool contains( const Object & x, BinaryNode \*t ) const {

if( t == nullptr )

return false;

else if( isLessThan( x, t->element ) )

return contains( x, t->left );

else if( isLessThan( t->element, x ) )

return contains( x, t->right );

else

return true; // tìm thấy

}

};

1. findMin và findMax

2 phương thức private tìm node có giá trị nhỏ nhất và lớn nhất.

/\*Phương thức private đệ qui tìm giá trị nhỏ nhất trên cây con t\*/

BinaryNode \* findMin( BinaryNode \*t ) const{

if( t == nullptr )

return nullptr;

if( t->left == nullptr )

return t;

return findMin( t->left );

}

/\*Phương thức private không đệ qui tìm giá trị nhỏ nhất trên cây con t\*/

BinaryNode \* findMax( BinaryNode \*t ) const{

if( t != nullptr )

while( t->right != nullptr )

t = t->right;

return t;

}

Nhiều lập trình viên có thể không dùng đệ qui mà dùng vòng lặp while như trong findMax: phải kiểm tra cây rỗng đầu tiên.

Chúng ta hoàn toàn có thể thay đổi con trỏ t trong thân hàm, vì đây chỉ là bản sao con trỏ từ đối số truyền vào mà thôi.

1. Insert

Trường hợp đúp giá trị có thể xử lí bằng cách thêm một trường trong node để ghi lại số lần xuất hiện, dù tốn chi phí nhưng tốt hơn là để xảy ra đúp, làm cây sâu hơn.

Và còn một vấn đề khi xóa: chúng ta giảm dữ liệu chỉ số lần xuất hiện xuống.

Dĩ nhiên, cách trên là không ổn nếu toán tử bé hơn (<) chỉ so sánh Key là một phần nhỏ trong dữ liệu lớn hơn. Trong trường hợp này, chúng ta giữ tất cả cấu trúc có cùng Key trong cùng một cấu trúc dữ liệu bổ sung, như list hay cây tìm kiếm khác.

**Phương thức đệ qui private chèn x vào cây con có node root là t.Và thiết lập lại node root cho cây con**

/\*Phiên bản dùng tham chiếu L\_value\*/

void insert( const Comparable & x, BinaryNode \* & t ){

if( t == nullptr )

t = new BinaryNode{ x, nullptr, nullptr };

else if( x < t->element )

insert( x, t->left );

else if( t->element < x )

insert( x, t->right );

else

; // Trùng giá trị;không làm gì cả

}

/\*Phiên bản dùng tham chiếu R\_value\*/

void insert( Comparable && x, BinaryNode \* & t ){

if( t == nullptr )

t = new BinaryNode{ std::move( x ), nullptr, nullptr };

else if( x < t->element )

insert( std::move( x ), t->left );

else if( t->element < x )

insert( std::move( x ), t->right );

else

; // Trùng giá trị;không làm gì cả

}

Chú ý khi node mới thêm vào, nó phải được gắn vào node cha. Nếu để tham số là \*t, sẽ không có ý nghĩa vì \*t chỉ là bản sao tham số truyền vào.

Vậy chúng ta phải truyền tham chiếu của con trỏ \*&t vào phương thức insert, để thay đổi bên trong hàm sẽ tác động lên đối số truyền vào.

1. Remove

Trường hợp node xóa tồn tại cả 2 cây con trái và phải, phương thức sẽ đưa node giá trị nhỏ nhất của cây con phải lên thay node xóa.

Sự không hiệu quả ở đây là việc tìm node và sau đó lại xóa node nhỏ nhất, tốn 2 lần duyệt cây con. Chúng ta triển khai phương thức **removeMin** để đơn giản vấn đề.

**Phương thức đệ qui private xóa x khỏi cây con có root là t.Và thiết lập node root mới cho cây convoid** remove( const Comparable & x, BinaryNode \* & t ) {

if( t == nullptr )

return; // Không tìm thấy, chẳng làm gì cả

if( x < t->element )

remove( x, t->left );

else if( t->element < x )

remove( x, t->right );

else if( t->left != nullptr && t->right != nullptr ) // Trường hợp có cả 2 cây con

{

t->element = findMin( t->right )->element;

remove( t->element, t->right );

}

else // Trường hợp có 1 cây con, đưa cây con lên thay

{

BinaryNode \*oldNode = t;

t = ( t->left != nullptr ) ? t->left : t->right;

delete oldNode;

}

}

Nếu số node phải xóa dự đoán là không nhiều, phương pháp phổ biến là **lazy deletion**:

* Phần tử cần xóa chỉ cần đánh dấu, và vẫn để nó trên cây.
* Cách này cũng hiệu quả khi trên cây tồn tại các node đúp. Khi này dữ liệu chỉ số lần xuất hiện chỉ cần giảm xuống.
* Tránh chi phí việc cấp phát lại khi chèn lại node đã xóa.

1. Hàm hủy và hàm dựng sao chép

Hàm hủy sẽ gọi phương thức private makeEmpty(), đệ qui xóa từng node con của cây t, sau đó xóa chính node t.

Cuối cùng, gán node root t = nullptr.

Hàm dựng sao chép mặc định khởi tạo root = nullptr, sau đó sao chép đối tượng rhs thông qua phương thức private clone().

~BinarySearchTree( ){makeEmpty( );}

/\*Phương thức xóa toàn cây con t\*/

void makeEmpty(BinaryNode \* & t ){

if( t != nullptr ){

makeEmpty( t->left );

makeEmpty( t->right );

delete t;

}

t = nullptr;

}

/\*Copy constructor\*/

BinarySearchTree( const BinarySearchTree & rhs ) : root{ nullptr }{root = clone( rhs.root );}

/\* Phương thức clone trả về bản sao cây t\*/

BinaryNode \* clone(BinaryNode \*t ) const{

if( t == nullptr )

return nullptr;

else

return new BinaryNode { t->element, clone( t->left ), clone( t->right ) };

}

1. Duyệt cây – Revisited

Nhắc lại, chúng ta có 3 cơ chế duyệt:

* Duyệt inorder: node con trái, node cha, node con phải
* Duyệt postorder: 2 node con, node cha
* Duyệt preorder: node cha, 2 node con
* Duyệt theo level: liệt kê các node có cùng level trước

Thời gian chạy là O(N), mỗi node được ghé qua 1 lần, và kiểm tra nullptr.

Thông tin trên cây nhị phân có tổ chức nên khá dễ để liệt kê theo thứ tự. Hàm đệ qui bên dưới làm việc này:

/\* In nội dung cây theo thứ tự sắp xếp \*/

void printTree( ostream & out = cout ) const {

if( isEmpty( ) )

out << "Empty tree" << endl;

else

printTree( root, out );

}

/\*Hàm nội bộ in cây t theo thứ tự sắp xếp \*/

void printTree( BinaryNode \*t, ostream & out ) const {

if( t != nullptr ) {

printTree( t->left, out );

out << t->element << endl;

printTree( t->right, out );

}

}

Thỉnh thoảng chúng ta duyệt post order, ví dụ khi tính chiều cao node, ưu tiên tính chiều cao cây con trước.

/\* Phương thức private tính chiều cao cây con tại root t.\*/

int height( BinaryNode \*t ){

if( t == nullptr )

return -1;

else

return 1 + max( height( t->left ), height( t->right ) );

}

Duyệt pre order hữu ích khi cần đánh chiều sâu cho node.

Phương pháp triển khai chung là không dùng các biến thừa, đặt trường hợp suy biến nullptr lên trên cùng. Gọi đệ qui với tham số là cây con trái hay phải. Code càng đơn giản thì càng ít lỗi.

Kiểu duyệt theo level không thể đệ qui mà dùng queue. Tất cả các node có độ sâu d được nạp vào queue hết trước khi tăng d lên, và duyệt tiếp.

1. Phân tích đánh giá

Ngoại trừ makeEmpty(), chúng ta kỳ vọng số lần duyệt cho mọi triển khai trên cây là O(logN) lần hay O(d) với d là độ sâu của node cần tìm.

Trong trường hợp remove, sẽ là node thay thế trong 1 trong 2 cây con.

Chúng ta sẽ chứng minh rằng độ sâu trung bình tất cả các node trên cây là O(logN) trên giả định rằng tất cả thao tác chèn như nhau.

* Xét cây T, gọi D(N) là tổng độ dài đường nội bộ trên cây T gồm N node. D(1) = 0.
* Gọi i là số node cây con trái, D(i) là độ dài đường nội bộ cây con trái của root. với 0 ≤ i < N
* Số node cây con phải: N-i-1.

Tại node root, tất cả node có độ sâu là 1, tức có N-1 đường nội bộ

Vậy ta có độ dài đường nội bộ N-node: D(N) = D(i) + D(N − i − 1) + N − 1

Nếu tất cả cây con có kích thước bằng nhau thì giá trị trung bình của cả D(i) và D(N − i − 1) là :

Suy ra:

Độ sâu kì vọng mọi node là O(logN).

Ví dụ khi tạo ngẫu nhiên 500 node. Độ sâu kì vọng là 9.98, tức thời gian trung bình mọi thao tác là O(logN), nhưng không thực sự là vậy.

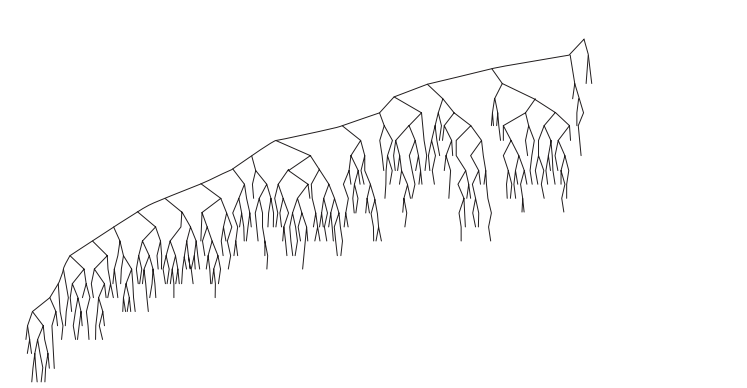
Các thao tác chèn, xóa có thể khiến cây không trong trạng thái cân bằng

Trong giải thuật xóa node, nếu đưa node Min cây con phải lên sẽ khiến cây con trái sâu hơn. (Ngược lại, đưa node Max cây con trái khiến cây con phải sâu hơn)



*Cây nhị phân tìm kiếm ngẫu nhiên 500 nodes.*

Không thể đánh giá chính xác độ ảnh hưởng trong thuật toán xóa node, nhưng người ta đã chỉ ra rằng nếu luân phiên chèn/xóa bình phương N lần, cây sẽ có độ sâu kì vọng là .



*Sau hàng triệu lần chèn/xóa, độ sâu trung bình : 12.51*

Nếu N đủ lớn, tới triệu lần, cây sẽ nặng về phía bên trái.

Chúng ta có thể triệt tiêu độ sai lệch bằng cách ngẫu nhiên chọn Max node cây con trái hay Min node cây con phải lên thay node xóa.

Không ai có thể kiểm chứng được cách trên đem lại mức độ cân bằng cho cây đến mức nào. Khó để thấy hiệu quả trên cây nhỏ.

Nhưng khá kì lạ là sau bình phương N cặp chèn/xóa, cây dường như trở nên cân bằng!

Khi Lazay deletion được dùng, mọi thao tác sẽ luôn là O(logN). Ngoại trừ trường hợp lạ lùng bên trên, kết quả luôn nhất quán với các hành vi.

1. Độ cân bằng

Nếu các giá trị không phải ngẫu nhiên, mà được sắp xếp trước, thì các thao tác chèn mất O(n) lần như trên DSLK, cây sẽ không có cây con trái hay phải.

Một giải pháp là định nghĩa điều kiện cân bằng cho cây: Không node nào được phép có độ sâu quá qui định.

Có khá nhiều giải thuật triển khai cây cân bằng, hầu hết sẽ phức tạp hơn so với cây nhị phân tìm kiếm, có số lần duyệt trung bình lâu hơn.

Chúng ta sẽ đề cập tới 2 cấu trúc:

* AVL Tree: cây cân bằng khá cổ điển.
* Slay Tree: cây mở rộng

Phương thức Slay Tree, không qui định điều kiện cân bằng như AVL, nên cây có độ sâu tùy ý, nhưng sau mỗi thao tác, luật tái cấu trúc được áp dụng để tối ưu cho về sau.

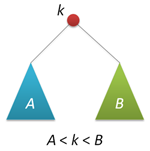
2 phương thức AVL và Slay Tree được xem là điển hình cho 2 trường phái tự cân bằng.

Không thể bảo đảm mọi thao tác là O(logN) trong trường hợp xấu nhất trên cây nhị phân tìm kiếm ngẫu nhiên. Nhưng vẫn có thể tối ưu cho chuỗi M thao tác mất tổng cộng O(M\*logN) lần duyệt với Slay Tree.

##### Cây AVL

AVL là kiểu đầu tiên của cây nhị phân tim kiếm cân bằng, phát minh bởi GM.Adelson-Velskii và E.M.Landis vào năm 1962.

1. Đánh giá cây AVL

Trước hết AVL là cây nhị phân tìm kiếm, vẫn tuân theo qui tắc : khóa một node luôn lớn hơn mọi khóa cây con trái và không nhỏ hơn mọi khóa cây con phải

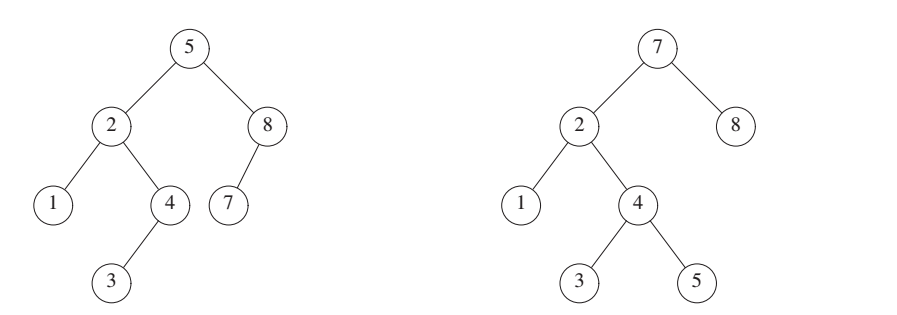
Khái nhiệm cân bằng trên cây AVL:

* Đảm bảo độ sâu luôn là O(logN)
* Với mọi node trên cây, chênh lệch chiều cao giữa hai cây con không quá 1.

Vấn đề hiệu suất khi lưu trữ thông tin chiều cao:

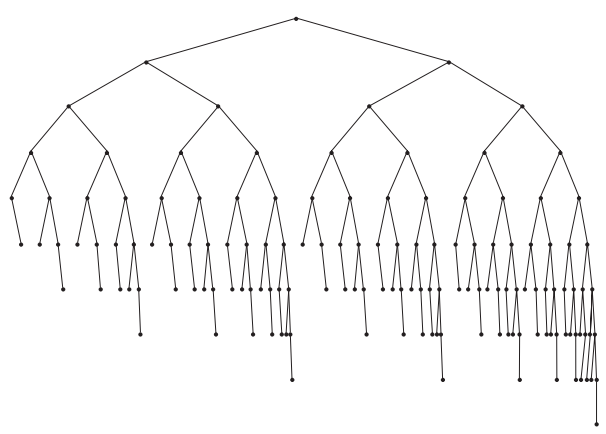
* Tránh phải tính toán lại chỉ số cân bằng, và giúp đơn giản hơn trong coding.
* Không ảnh hưởng nhiều tới chi phí lưu trữ, chỉ 8-bit

Gọi h là chiều cao cây AVL gồm n node: có thể chỉ ra rằng h chỉ nằm trong khoảng log2(N + 1) tới 1.44\*log2(N + 2) − 1.328



*Cây bên trái hình là AVL, cây bên phải thì không do chiều cao cây con trái > 2 so với chiều cao cây con phải của* node 7

Trên thực tế thì h sẽ không lớn hơn logN nhiều.



*Cây AVL 143 node, có chiều cao là 9*

Xem ví dụ cây AVL trong hình, cây con trái có chiều cao là 7, trong khi cây con phải là 8.

Điều này chỉ ra số node tối thiểu của cây có chiều cao h:

S(h) = S(h − 1) + S(h − 2) + 1

* h = 0: S(h) = 1
* h = 1: S(h) = 2

Hàm S(h) nhìn khá giống dãy số Fibonacci, có giới hạn là chiều cao của cây.

Thao tác chèn node có thể dẫn tới vi phạm luật cân bằng, nên chúng ta cần cân bằng lại tất cả node trên đường về root, vì chỉ những node này có cây con bị thay đổi.

1. Cấu trúc node

Để đơn giản, chúng ta định nghĩa khóa là kiểu số, và là duy nhất.

Cấu trúc node cây AVL:

struct node{

int key;

unsigned char height;

node\* left;

node\* right;

node(int k) { key = k; left = right = 0; height = 0; }

};

Trường key chứa giá trị khóa, height chỉ chiều cao của node. Hàm dựng cơ bản để khởi tạo giá trị

Theo nguyên tắc, node cây AVL không chứa thông tin chiều cao.

Chiều cao cây: h < 1.44\*log2(n + 2) − 0.328, nghĩa là nếu số node = 109 (1 tỉ node chiếm hơn 10GB vùng nhớ cấp phát), thì chiều cao cây sẽ không tới 44, tức không tới 1-byte. Do đó, sẽ không làm tăng vùng nhớ bao nhiêu. Mặt khác còn giúp triển khai cây dễ dàng hơn.

1. Chỉ số cân bằng và chiều cao

Nguyên tắc chung mà chúng ta thỏa thuận trước là:

* Chỉ số cần bằng một node: chiều cao cây con phải trừ chiều cao cây con trái.
* Chiều cao một node: tính từ node lá sâu nhất, hay bằng chiều cao một trong hai cây con sâu hơn + 1
* Chiều cao node rỗng sẽ là -1
* Chỉ số cân bằng chỉ chấp nhận giá trị : -1, 0 và 1

Thứ nhất là hàm lấy chiều cao node:

unsigned char get\_height(node\* p\_tree){

return p\_tree ? p\_tree ->height : -1;

}

Hàm bfactor() để tính chỉ số cân bằng của một node:

int bfactor(node\* p\_tree){

return get\_height(p\_tree ->right) - get\_height(p\_tree ->left);

}

Hàm tính lại giá trị đúng chiều cao của một node (với điều kiện chỉ số cân bằng cây con trái và phải đã chính xác)

void fix\_height(node\* p\_tree){

unsigned char h\_left = get\_height(p\_tree->left);

unsigned char h\_right = get\_height(p\_tree->right);

p\_tree->height = (h\_left > h\_right? h\_left: h\_right) + 1;

}

Các hàm trên không phải đệ qui nên thời gian thuật toán là О(1).

1. Trường hợp mất cân bằng

Xét node α, chỉ có 4 trường hợp chèn node sau làm cây α mất tính cân bằng:

1. Chèn vào cây con trái của node trái α

2. Chèn vào cây con phải của node trái α

3. Chèn vào cây con trái của node phải α

4. Chèn vào cây con phải của node phải α

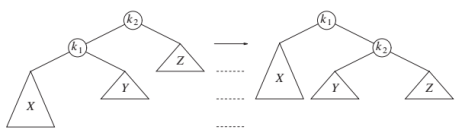
Trường hợp 1 và 4 là tương tự với α, cũng như trường hợp 3 và 4. Về học thuyết có hai trường hợp cơ bản, về lập trình sẽ là 4 trường hợp.

Trường hợp cơ bản 1: việc chèn node xảy ra bên ngoài (left-left hay right-right), được tái cân bằng qua phép xoay đơn.

Trường hơp cơ bản 2: việc chèn node bên trong (left-right hay right-left), được tái cân bằng qua phép xoay kép.

Trong phần nâng cao về cấu trúc dữ liệu, bạn sẽ học về phương pháp cây cân bằng khác có sự giám sát chặt chẽ hơn trong kỹ thuật triển khai.

1. Phép xoay đơn

Hình dưới minh họa phép xoay đơn cân bằng trường hợp chèn left-left: (dấu gạch đứt thể hiện độ sâu).

Sau khi chèn node vào cây con trái k1, cây con X sâu thêm một bậc, chênh lệnh 2 bậc so với cây Z.

Cây Y không thể cùng bậc với X sau khi chèn vì nếu vậy, k2 đã mất cân bằng trước đó.

Hình : phép xoay đơn trường hợp chèn left-left

Và cây Y cũng không thể cùng bậc với cây Z, vì nếu vậy k1 là node đầu tiên đi tới root vi phạm nguyên tắc cân bằng.

Giải pháp: Dùng phép xoay phải tại quanh node k2

* Chúng ta phải chuyển cây X lên một bậc và cây Z xuống một bậc.
* K2 > k1, nên k2 sẽ là node phải của k1.
* Cây X, Z giữ nguyên vị trí. Riêng cây Y sẽ gắn vào cây con trái của k2 để thỏa mãn yêu cầu thứ tự cây nhị phân

Kết quả là cây nhị phân thỏa mãn AVL:

* X lên một bậc, Y giữ bậc, Z xuống một bậc.
* k1 và k2 đều có hai cây con cùng chiều cao.
* Cây nhị phân mới vẫn giữ được chiều cao sau khi chèn
* K1 và k2 đều thay đổi chiều cao, nên phải tính lại

Do đó không cần cập nhật lại chiều cao cho đường tới root.

Triển khai code xoay phải

/\* Xoay phải tại p\_tree\*/

node\* rotate\_right(node\* p\_tree) {

if (p\_tree == NULL) return NULL;

//lấy node trái p\_tree

node\* p\_subtree\_left = p\_tree->left;

// Đặt cây con phải p\_subtree\_left làm cây con trái của p\_tree

p\_tree->left = p\_subtree\_left->right;

// Đặt p\_tree làm cây con phải của p\_subtree\_left

p\_subtree\_left->right = p\_tree;

// Tính lại chiều cao p\_tree và p\_subtree\_left

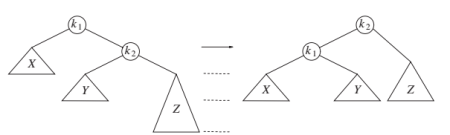
fix\_height(p\_tree);

fix\_height(p\_subtree\_left);

// trả về node root mới

return p\_subtree\_left;

}



Trường hợp thứ 4 cũng tương tự, hình bên cạnh minh họa phép xoay trái.

Hình : phép xoay trái trường hợp chèn right-right

Xoay trái tương tự như xoay phải:

/\*Xoay trái tại p\_tree\*/

node\* rotate\_left(node\* p\_tree) {

if (p\_tree == NULL) return NULL;

//lấy node phải p\_tree

node\* p\_subtree\_right = p\_tree->right;

//Đặt cây con trái p\_subtree\_right vào cây con phải p\_tree

p\_tree->right = p\_subtree\_right->left;

//Đặt p\_tree vào cây con trái p\_subtree\_right

p\_subtree\_right->left = p\_tree;

// Tính lại chiều cao p\_tree và p\_subtree\_right

fix\_height(p\_tree);

fix\_height(p\_subtree\_right);

//trả về node root mới

return p\_subtree\_right;

}

Ví dụ bên dưới mô tả cây phân khi chèn các node 3, 2, 1, và 4 tới 7 theo thứ tự.

|  |
| --- |
| Hình : mất cân bằng tại root, xoay phải quanh node root |
| Hình :mất cân bằng tại node 3, xoay trái quanh node 3 |
| Hình : mất cân bằng tại root sau khi thêm node 6. Xoay trái quanh root. |
| Hình 6: mất cân bằng tại node 5 sau khi thêm node 7, xoay trái quanh node 5 |

1. Phép xoay kép

Giải thuật xoay đơn sẽ không giải quyết được trường hợp hai và ba. Vấn đề trong hình số 7 bên dưới là cây Y sau khi chèn trở nên quá sâu, phép xoay đơn không khiến cây cân bằng lại được.

|  |
| --- |
| Hình : Xoay đơn right không giải quyết được trường hợp thứ 2 |
| Hình : Dùng xoay kép left-right để tái cân bằng    Thực tế là cây con Y đã được chèn, nên nó không thể rỗng. |

Theo hình minh họa số 8, giả sử rằng cây Y gồm một node root k2 và hai cây con B và C. Như vậy cây của chúng ta có tổng cộng 4 cây con (A, B, C, D) nối với 3 node (k1, k2, k3).

Chắc chắn là một trong hai cây B và C sẽ lệnh 2 bậc so với cây D (nếu chúng không rỗng), nhưng chúng ta không thể biết chắc là cây nào.

Trong hình 8, để đơn giản hóa, cả B và C được vẽ lệch 3/2 bậc so với D. Để tái cân bằng, không thể để K3 là root được, vì phép xoay phải hình 7 bị mất cân bằng tại k1.

Giải pháp là để K2 là root mới: đẩy k1 thành node trái của k2 và k3 thành node phải của k2. Dễ dàng thấy là cấu trúc cây sau đó thỏa AVL, cũng giống như xoay đơn, lần xoay kép này đã giữ được chiều cao của cây như trước khi chèn.

Tương tự cho trường hợp 3:

|  |
| --- |
| Hình : xoay kép right-left cân bằng cho trường hợp 3 |

Ví dụ:

|  |
| --- |
| Chúng ta dùng ví dụ trước đó, chèn thêm node 16, 15, 14, 13, 12, 11, 10.  Khi chèn node 16, cây chưa mất tính cân bằng. Khi node 15 được chèn, cây mất cân bằng tại node 7.  Trường hợp này, k1 là node 7, k2 là node 15, k3 là node 16. Cây con A, B, C, D rỗng.  Chèn node theo trường hợp số 3, tiến hành xoay kép: xoay phải tại k3 –xoay trái tại k1. |
| Chèn tiếp node 14, mất cân bằng tại node 6 theo trường hợp số 3: k1 là node 6, k3 là node 15, k2 là node 7.  Cây con B rỗng, cây C là node 14, D là node 16, cây A là node 15.  Tiến hành xoay kép: xoay phải tại k3 – xoay trái tại k1. |
| Node 13 chèn chỉ làm cây mất cân bằng tại root, vì 13 không nằm trong khoảng giữa 4 và 7, nên ta biết chỉ cần dùng phép xoay trái tại node 4 là được. |
| Chèn node 12, xoay phải tại node 14. |
| Chèn node 11, xoay phải tại node 15. |
| Cuối cùng chèn node 9, gây mất cân bằng tại node 10. Vì 9 nằm giữa 10 và 8 (node con 8 của 10 nằm trên đường đi tới node 9).  Chúng ta thực hiện xoay kép: xoay trái tại node 8 – xoay phải tại node 10. |

1. Hàm cân bằng node

Mục tiêu:

* Tính lại chiều cao node p\_tree
* Kiểm tra chỉ số cân bằng node : nếu nằm trong khoảng (-1, 1), cây không mất cân bằng, trả về p\_tree. Nếu bằng 2 hay -2, cây mất cân bằng
* Bfactor() = 2: kiểm tra chỉ số cân bằng cây con phải p\_tree, dùng thao tác xoay trái hay xoay kép phải – trái để tái cân bằng. Trả về node root mới
* Bfactor() = -2: kiểm tra chỉ số cân bằng cây con trái p\_tree, dùng thao tác xoay phải hay xoay kép trái – phải để tái cân bằng. Trả về node root mới

Triển khai:

Đoạn code kiểm tra điều kiện trước khi thao tác xoay cây:

/\*cân bằng cho node p\*/

node\* balance(node\* p\_tree) {

//tính lại chiều cao node

fix\_height(p\_tree);

//trường hợp p\_tree lệch phải

if (bfactor(p\_tree) == 2) {

//xét cây con phải

if (bfactor(p\_tree->right) > 0) {//lệch phải - xoay trái

return rotate\_left(p\_tree);

}

if (bfactor(p\_tree->right) < 0) {//lệch trái - xoay phải - trái

p\_tree->right = rotate\_right(p\_tree->right);

return rotate\_left(p\_tree);

}

}

//trường hợp p\_tree lệch trái

if (bfactor(p\_tree) == -2) {

//xét cây con trái

if (bfactor(p\_tree->left) < 0) {//lệch phải - xoay phải

return rotate\_right(p\_tree);

}

if (bfactor(p\_tree->left) > 0) {//lệch phải - xoay trái - phải

p\_tree->left = rotate\_left(p\_tree->left);

return rotate\_right(p\_tree);

}

}

// trường hợp p\_tree đã cân bằng rồi

return p\_tree;

}

Các hàm xoay đơn và cân bằng node không dùng vòng lặp hay đệ qui, nên thời gian chạy là hằng số, bất kể kích thước cây AVL.

1. Chèn key

Khi chèn node có giá trị X vào cây AVL T, chúng ta đệ qui hàm chèn X vào cây con tương ứng của T (gọi nó là TLR). Nếu chiều cao của TLR không thay đổi, coi như xong.

Ngược lại, T mất cân bằng, chúng ta làm xoay đơn hay kép tùy theo X và giá trị cây T và TLR, cập nhật lại chiều cao cho phần còn lại của cây.

Vì một thao tác xoay luôn đủ đáp ứng yêu cầu, một phiên bản code không đệ qui nói chung là sẽ nhanh hơn đệ qui, nhưng trên trình biên dịch hiện đại, sự khác biệt không còn quan trọng như trước kia. Thêm nữa, hàm không đệ qui khá khó để viết chính xác, khi mà đệ qui hàm lại ngắn gọn, dễ hiểu hơn.

Giải thuật chèn key cũng tương tự như trên cây nhị phân tìm kiếm. Chỉ khác là chúng ta trả về node sau khi cân bằng.

Triển khai:

// Chèn khóa k vào cây p

node\* insert(node\* p, int k){

if( !p ) return new node(k);

if( k<p->key )

p->left = insert(p->left,k);

else

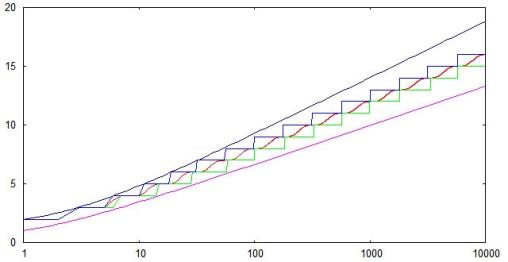
p->right = insert(p->right,k);

return balance(p);

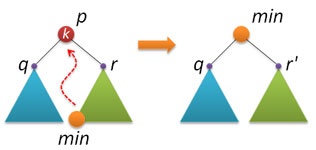
}

Để kiểm tra thuật toán chèn: chúng ta thử nghiệm trên dãy số ngẫu nhiên từ 1 tới 1000. Các số được chèn ngẫu nhiên trên cây AVL rỗng và chiều cao của cây được tính toán sau mỗi lần chèn.

Kết quả thu được tính trung bình cho 1000. Biểu đồ sau minh họa tính lệ thuộc vào số phần tử n cho chiều cao trung bình của cây (màu đỏ), chiều cao tối thiểu (màu xanh), chiều cao tối đa (màu xanh dương).

Thực nghiệm chứng minh rằng chiều cao từ các giá trị ngẫu nhiên luôn nằm trong khoảng giới hạn lý thuyết. Trong điều kiện lý tưởng (dãy sắp xếp tăng dần), chiều cao cây đạt được tối thiểu.

1. Xóa Key



Theo lý thuyết chung, ta tìm node có giá trị key là k (nếu không tồn tại thì chẳng làm gì cả).

Nếu tìm thấy: chúng ta tìm node bé nhất trên cây con trái node đó và đưa node đó lên thay thế node bị xóa.

Một vài điều cần lưu ý:

* Nếu node tìm thấy không có cây con phải, theo thuộc tính cây AVL, node này sẽ chỉ có một node trên cây con trái. Chúng ta chỉ cần xóa node và trả về con trỏ tới cây con trái.
* Nếu node tìm thấy có cây con phải, tìm giá trị nhỏ nhất. Theo cây nhị phân, chúng ta dùng triển khai sau:

/\*kiếm node nhỏ nhất trong p\_tree \*/

node\* find\_min\_node(node\* p\_tree) {

//trường hợp này không thể xảy ra

if (p\_tree == NULL) return NULL;

//trường hợp tìm thấy

if (p\_tree->left == NULL) {

return p\_tree;

}

//trường hợp chưa tìm thấy, đệ qui hàm

return find\_min\_node(p\_tree->left);

}

Hàm tiếp theo để xóa node giá trị nhỏ nhất. Theo thuộc tính cây AVL, chỉ có hai trường hợp xảy ra:

* Node nhỏ nhất là node lá
* Chỉ có duy nhất một node trên cây con trái của node nhỏ nhất

Triển khai hàm tương tư như cây nhị phân tìm kiếm, node trả về phải balance :

// xóa node nhỏ nhất trong p\_tree

node\* remove\_min\_node(node\* p\_tree, node\* p\_min\_node) {

//Trường hợp này không thể xảy ra

if (p\_tree == NULL) return NULL;

//Trường hợp tìm thấy

if (p\_tree == p\_min\_node) {

//Trả về cây con phai p\_min\_node

return p\_min\_node->right;

}

//Trường hợp chưa tìm thấy, đệ qui hàm

p\_tree->left = remove\_min\_node(p\_tree->left, p\_min\_node);

return balance(p\_tree);

}

Hàm triển khai xóa node tương tư như cây nhị phân tìm kiếm, chỉ khác là bạn phải nhớ balance cho node trả về:

node\* remove(node\* p\_tree, int key) {

//Trường hợp không tìm thấy

if (p\_tree == NULL) return NULL;

//Trường hợp tìm thấy

if (p\_tree->key == key) {

//Trường hợp node xóa có một cây con hay không có cây con nào

if (p\_tree->left == NULL) {

node\* p\_right\_subtree = p\_tree->right;

delete p\_tree;

return p\_right\_subtree;

}

if (p\_tree->right == NULL) {

node\* p\_left\_subtree = p\_tree->left;

delete p\_tree;

return p\_left\_subtree;

}

//Trường hợp node xóa có hai cây con

//tìm min node từ cây con phải node xóa

node\* p\_min\_node = find\_min\_node(p\_tree->right);

//Tái cấu trúc cây, đưa p\_min lên thay node xóa

p\_min\_node->right = remove\_min\_node(p\_tree->right, p\_min\_node);

p\_min\_node->left = p\_tree->left;

delete p\_tree;

//Trả về p\_min đã balance

return balance(p\_min\_node);

}

//Trường hợp chưa tìm thấy, đệ qui hàm

if (key < p\_tree->key) {

p\_tree->left = remove(p\_tree->left, key);

} else {

p\_tree->right = remove(p\_tree->right, key);

}

//Trả về p\_tree đã balance

return balance(p\_tree);

}

1. template version

template <typename Comparable>

class AVLTree{

struct AvlNode{

Comparable element;

AvlNode \*left;

AvlNode \*right;

int height;

AvlNode( const Comparable & ele, AvlNode \*lt, AvlNode \*rt, int h = 0 )

: element{ ele }, left{ lt }, right{ rt }, height{ h } { }

AvlNode( Comparable && ele, AvlNode \*lt, AvlNode \*rt, int h = 0 )

: element{ std::move( ele ) }, left{ lt }, right{ rt }, height{ h } { }

};

/\*Hàm lấy chiều cao node t : -1 nếu t = nullptr \*/

int height( AvlNode \*t ) const{

return t == nullptr ? -1 : t->height;

}

/\*Hàm chèn node\*/

void insert( const Comparable & x, AvlNode \* & t ){

if( t == nullptr )

t = new AvlNode{ x, nullptr, nullptr };

else if( x < t->element )

insert( x, t->left );

else if( t->element < x )

insert( x, t->right );

balance( t );

}

static const int ALLOWED\_IMBALANCE = 1;

void balance( AvlNode \* & t ){

if( t == nullptr )

return;

if( height( t->left ) - height( t->right ) > ALLOWED\_IMBALANCE )

if( height( t->left->left ) >= height( t->left->right ) )

rotateWithLeftChild( t );

else

doubleWithLeftChild( t );

else

if( height( t->right ) - height( t->left ) > ALLOWED\_IMBALANCE )

if( height( t->right->right ) >= height( t->right->left ) )

rotateWithRightChild( t );

else

doubleWithRightChild( t );

t->height = max( height( t->left ), height( t->right ) ) + 1;

}

/\*\*

\* Rotate binary tree node with left child.

\* For AVL trees, this is a single rotation for case 1.

\* Update heights, then set new root.

\*/

void rotateWithLeftChild( AvlNode \* & k2 ) {

AvlNode \*k1 = k2->left;

k2->left = k1->right;

k1->right = k2;

k2->height = max( height( k2->left ), height( k2->right ) ) + 1;

k1->height = max( height( k1->left ), k2->height ) + 1;

k2 = k1;

}

/\*\*

\* Double rotate binary tree node: first left child

\* with its right child; then node k3 with new left child.

\* For AVL trees, this is a double rotation for case 2.

\* Update heights, then set new root.

\*/

void doubleWithLeftChild( AvlNode \* & k3 ) {

rotateWithRightChild( k3->left );

rotateWithLeftChild( k3 );

}

/\*Hàm xóa item x trên cây t. Thiết lập lại root mới cho cây\*/

void remove( const Comparable & x, AvlNode \* & t ) {

if( t == nullptr )

return; // Không tìm thấy x, return

if( x < t->element )

remove( x, t->left );

else if( t->element < x )

remove( x, t->right );

else if( t->left != nullptr && t->right != nullptr ) // Trường hợp có 2 cây con

{

t->element = findMin( t->right )->element;

remove( t->element, t->right );

}

else

{

AvlNode \*oldNode = t;

t = ( t->left != nullptr ) ? t->left : t->right;

delete oldNode;

}

balance( t );

}

};

##### Cây mở rộng (Slay Tree)

1. Giới thiệu

Splay Tree được giới thiệu bởi Sleator và Tarjan vào năm 1985 . Cấu trúc Splay Tree thuộc về lớp các cây tìm kiếm nhị phân tự cân bằng (Self-Adjusting Binary Search Tree - BST), có nghĩa là nó sẽ tự điều chỉnh cấu trúc của nó mỗi khi có một thao tác thực hiên trên cây.

So với cây cân bằng AVL Tree và Red Black Tree, Splay Tree có một số ưu điểm sau:

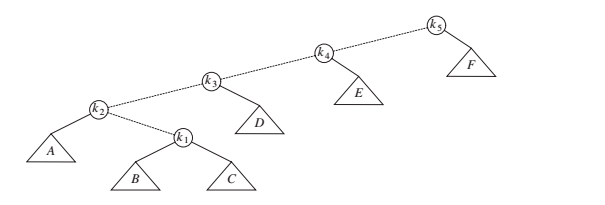
* Dễ thực thi hơn
* Cần ít bộ nhớ hơn do không cần duy trì thêm thông tin để cân bằng.
* Hiệu quả hơn nhiều nếu như phân phối truy nhập các khóa là không đồng nhất (skew). Cụ thể, nếu một khóa được truy nhập liện tục nhiều lần thì lần truy nhập đầu tiên mất thời gian ( khấu trừ) O(logN) còn các lần sau đó là O(1).

Splay tree tối ưu dựa vào mức độ truy xuất của node, dùng phép xoay cây của AVL tree để cấu trúc lại cây.

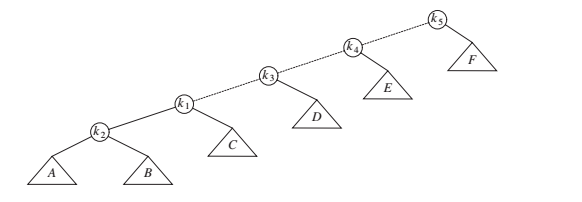
Tuy nhiên, điểm yếu của Splay Tree là chiều cao của cây có thể là N, và một số thao tác có thể rất tốn kém (nhưng trung bình sẽ là O(logN).

1. Xoay đơn

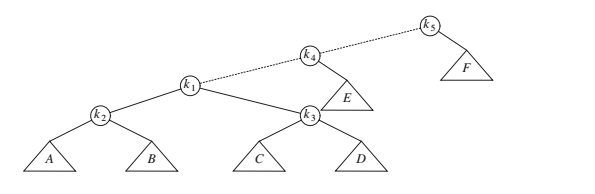
Tái cấu trúc cây theo phép xoay đơn đi từ dưới lên: tức xoay mọi node trên đường truy xuất với node cha. Hình dung phép xoay khi truy xuất node k1 (trong thao tác tìm):



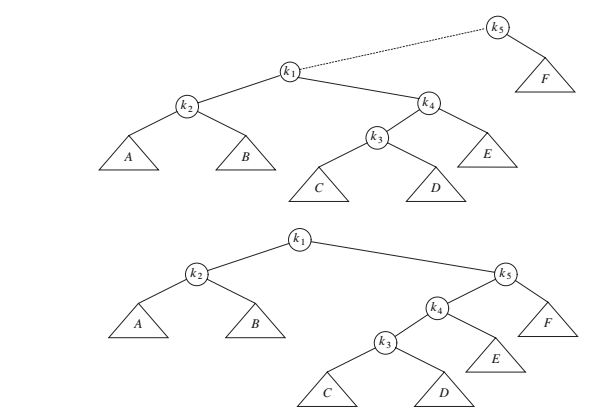
Đường truy xuất gạch đứt. K1 xoay đơn với node cha K2:



Xoay k1 lên k3:



Tiếp tục tới khi xoay k1 về root:



Quá trình này sẽ đẩy node k1 về root theo đường truy xuất, lần truy xuất k1 trong tương lai sẽ nhanh hơn. Nhưng đồng thời nó mở rộng cây con phải của k1, các node trên đường truy xuất bị đẩy xuống.

Phép xoay giúp k1 dễ truy xuất hơn nhưng không cải thiện cho các node trên đường truy xuất: chưa tiệm cận tới hiệu suất O(logN) được.

Tình huống xấu nhất khi chèn key 1, 2, 3…, N thời gian tạo cây là O(N) tổng cộng, và chỉ có cây con trái. Phần xấu nhất là khi truy xuất node key = 1, phải duyệt N phần tử, sau khi xoay 1 lên root. key 2 cũng mất N, key 3 mất N-1 …

Tổng thời gian truy xuất tất cả key theo thứ tự là :

Sau khi truy xuất, cây trở về trạng thái bạn đầu và cứ lặp lại tuần tự như vậy.

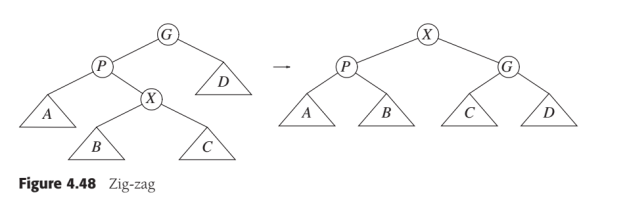
1. Xoay mở rộng (Splay)

Ý tưởng phép Splay tương tự như xoay đơn, nhưng có chọn lọc – 2 bước - trong thao tác xoay. Chiều xoay vẫn giữ từ dưới lên (bottom up) theo đường truy xuất.

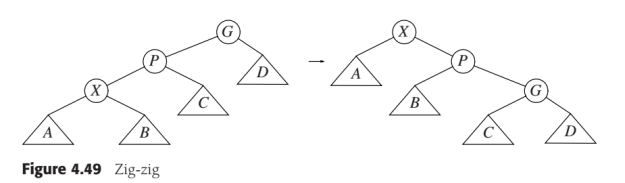
Gọi X là node truy xuất. Xét X có node cha là root, thì chỉ cần xoay X về root, đây là phép xoay cuối cùng trên đường truy xuất.

Xét trường hợp X không gần root. Gọi P là node cha của X và G là node cha của P, sẽ xảy ra 2 trường hợp (Xét đối xứng qua cũng sẽ tương tự):

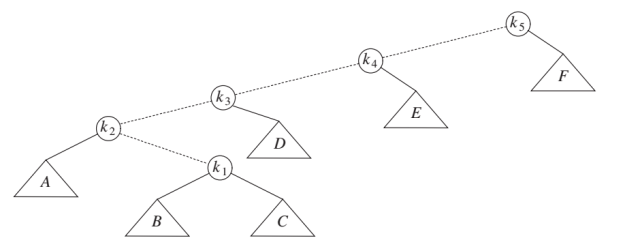
* Dạng Zig – zag: X là node con phải của P, chúng ta làm phép xoay kép tương tự phép xoay AVL



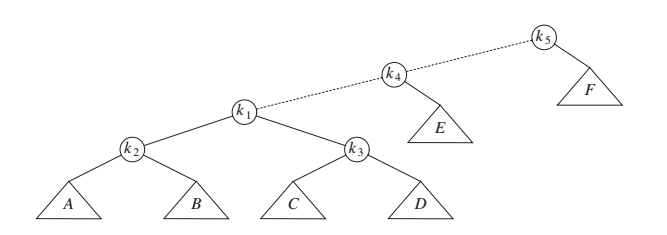
* Dạng Zig-zig: X là node con trái của P (X, P đều nằm trên cây con trái của G), chúng ta xoay cây tính từ X tới G về bên phải, tức xoay phải giữa P và G trước, sau đó tới X và P.



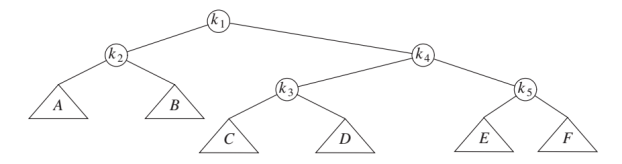
Xét lại node k1 trong ví dụ bài trước:



Bước 1: zig-zag tại k1, xoay kép AVL giữa k1,k2 và k3:



Bước 2: zig-zig tại k1, xoay k1, k4, và k5:



Hơi khó hình dung, nhưng phép splay không chỉ di chuyển node về root, mà làm sâu hầu hết node trên đường về root về một phía cây con của node truy xuất.

Although it is hard to see from small examples, splaying not only moves the accessed

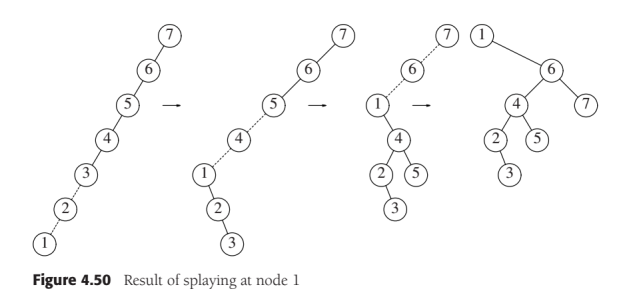
node to the root but also has the effect of roughly halving the depth of most nodes on the

access path (some shallow nodes are pushed down at most two levels).

1. Đánh giá Splay

Khi so sánh splay với phép xoay đơn, chúng ta xét lại trường hợp chèn key 1, 2, 3, . . . , N liên tiếp trên cây rỗng.

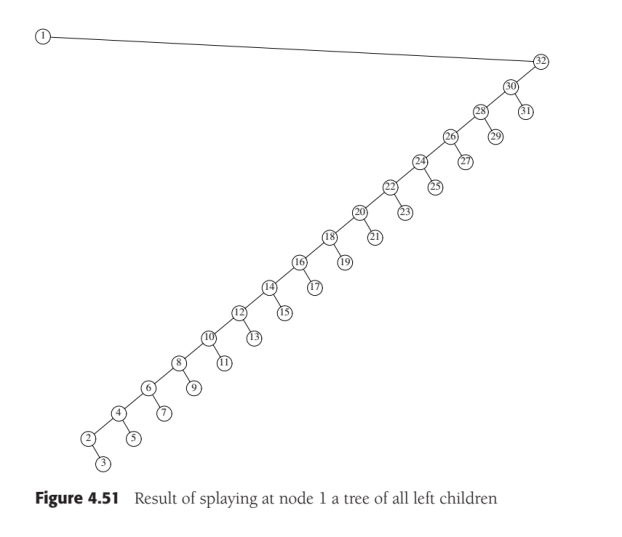
Bước đầu xây cây vẫn mất O(N) cho cả 2 (Khi chèn có dùng phép xoay).

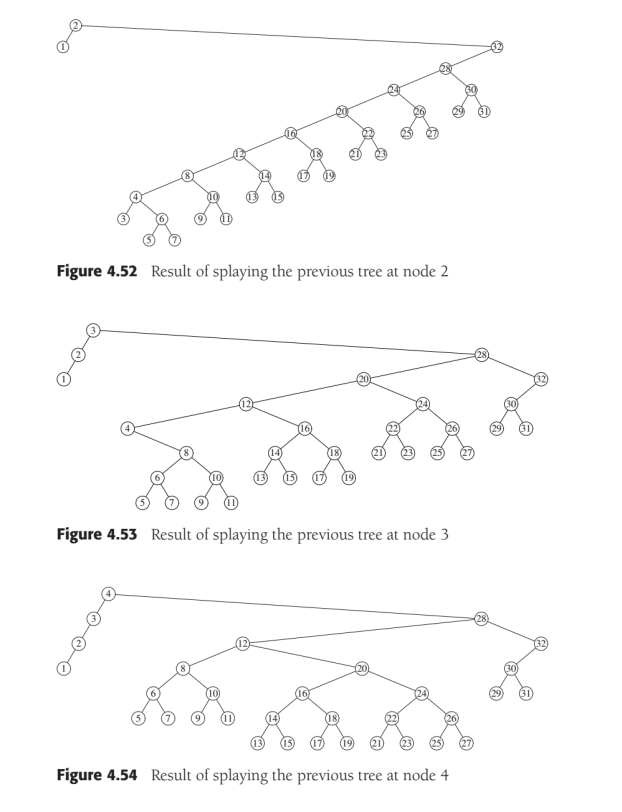
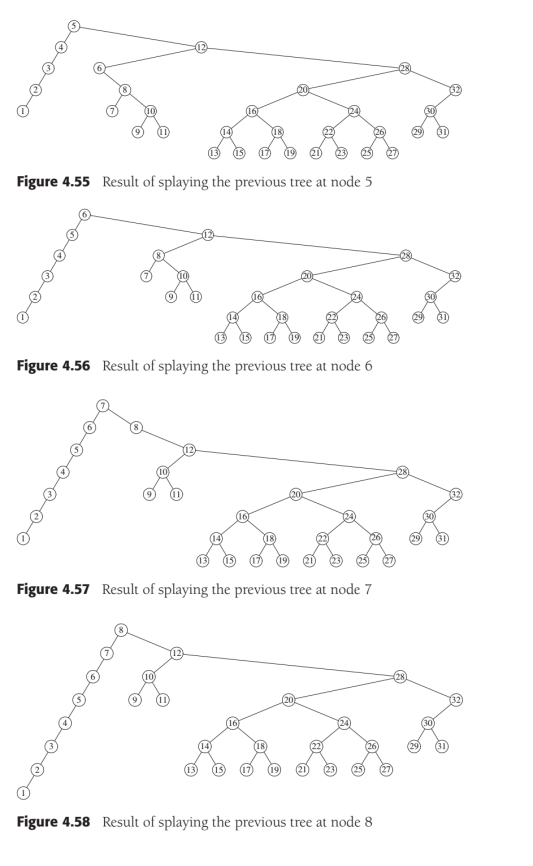
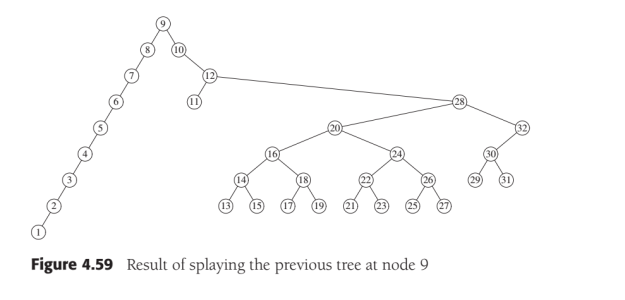


Hình trên mô tả phép xoay trên key 1. Điều khác biệt là key 1 mất N, key 2 chỉ mất N/2, cây con của key 1 không sâu như trước.

Các lần truy xuất sau sẽ tiệm cận tổng thời gian về gần logN. N = 7 là quá nhỏ để thấy hiệu ứng này.

Hình dưới là kết quả truy xuất key 1 tới 9 trên cây 32 node chỉ gồm cây con trái, các khuyết điểm của xoay đơn được triệt tiêu



Khi đường truy xuất dài, dẫn tới thời gian tìm lâu hơn bình thường, phép xoay có khuynh hướng cải thiện về sau. Khi truy xuất trở nên dễ, phép xoay không tốt cũng không xấu.

Phân tích splay tree rất khó, nhưng qua thực nghiệm, triển khai lại đơn giản hơn hầu hết các cây cân bằng.

Có rất nhiều mô hình splay tree triển khai tốt trên thực tế.

##### B-Tree

1. Khả năng lưu trữ đĩa cứng

Khi cấu trúc dữ liệu cực lớn, không thể chứa trong main memory được (RAM), mà lưu trữ trên đĩa cứng. Lúc này hiệu suất số Big-O không còn ý nghĩa nhiều nữa.

Vấn đề chính nằm ở thời gian truy xuất đĩa cứng, trong 1 giây, máy tính có thể xử lí hàng tỉ chỉ thị. Trong khi đó, đĩa cứng hoạt động dựa vào cơ học, nó tốn thời gian để quay đĩa và di chuyển đầu đọc đến đúng vị trí.

1. Tốc độ truy xuất

Giả sử tốc độ quay là 7,200 vòng/phút, tương đương 120 vòng/s.

1 vòng quay mất xấp xỉ 8.3 ms, trung bình mất nửa vòng quay để tìm đúng dữ liệu, nếu tính thêm thời gian di chuyển đầu đọc, chúng ta mất gần 8.3 ms (thời gian truy xuất ước tính là 9-11ms, là khá đúng)

Một sự thật rõ ràng là tốc độ truy xuất đĩa cứng quá chậm so với CPU, phải tính toán cẩn thận để tiết kiệm thời gian truy xuất đĩa, truy xuất càng nhiều thì chương trình càng chạy chậm.

Xét 10 triệu khóa 32-bytes là dữ liệu lái xe ở bang Florida. Cây nhị phân triển khai trên đĩa cứng và có nhiều user truy xuất trên hệ thống (tổng cộng 20 user).

Phân phối trung bình cho 20 user, chúng ta chỉ có khoảng hàng triệu chỉ thị xử lí trong 1s hay 120/20 = 6 lần truy xuất đĩa.

Cây nhị phân không cân bằng là 1 thảm họa rõ ràng, trường hợp xấu nhất là 10 triệu truy xuất đĩa.

Xét hiệu suất trung bình 1.38 logN, log của 10 triệu là 24, tính ra mất 32 truy xuất trung bình hay 5s.

Trên cây ngẫu nhiên, một Node có thể sâu gấp 3 lần so với Node nông nhất, nó tốn 100 lần truy xuất hay 16 giây.

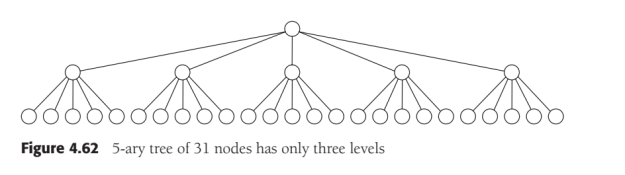
Cây ALV tốt hơn, trường hợp xấu nhất là 1.44 logN, tương ứng 25 lần truy xuất trung bình, hay 4 giây.

Chúng ta cần cải thiện số lần truy xuất là một hằng số nhỏ, như 3 hay 4. Code sẽ phức tạp và dài hơn, nhưng không thành vấn đề vì CPU xử lí rất nhanh.

1. Cây tẻ nhánh

Giải pháp là phải làm giảm chiều cao cây xuống hay tẽ nhánh nhiều hơn.

Một cây nhị phân cân bằng hoàn hảo 31 node có 5 cấp. Nhưng một cây 5 nhánh 31 Node, chỉ có 3 cấp.



Cây tẻ M-nhánh sẽ có đối đa M nhánh trên mỗi Node, nhánh càng nhiều thì chiều cao càng giảm.

Cây nhị phân là trường hợp tẻ 2-nhánh, chiều cao khoảng log2(N) và cần 1 khóa để xác định 2 cây con

Cây M-nhánh sẽ có chiều cao xấp xỉ logM(N), cần M-1 khóa để xác định M cây con.

Để đạt hiệu suất tốt, phải đảm bảo cây M-nhánh cân bằng theo nhiều cách. Trường hợp xấu nhất vẫn là danh sách liên kết đơn, do đó không thể triển khai như kiểu cây nhị phân ngẫu nhiên

Có một cách triển khai là dùng B-tree.

1. Đặc tính cân bằng cho B-tree

Theo lý thuyết, B-tree đảm bảo giới hạn số lần truy xuất đĩa. Một cây B gồm M thứ tự là cây M-nhánh với các đặc tính:

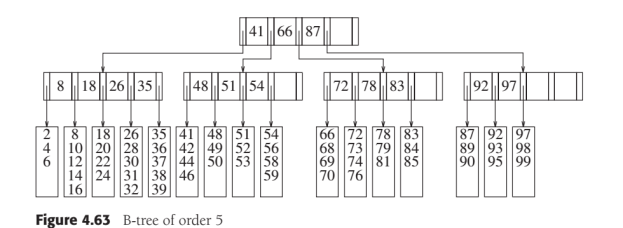
1. Dữ liệu chỉ chứa trên Node lá
2. Node không phải lá chứa M-1 khóa cho việc tìm kiếm, khóa “i” đại diện cho khóa nhỏ nhất trên cây con “i+1”
3. Root cũng có thể là lá hay giữa 2 hay M Node con
4. Tất cả Node không phải lá (ngoại trừ root) có từ M/2 tới M nhánh con
5. Tất cả Node lá có cùng độ sâu và có từ L/2 tới L dữ liệu.

Cách xác định số L sẽ được phân tích sau.

Ví dụ về B-tree 5 nhánh, để ý các Node không phải lá có từ 3 tới 5 Node con ( hay từ 2 tới 4 khóa). Node root chỉ có 2 Node con

Ở đây số dữ liệu của 1 Node lá được qui định là L = 5. (L = M-nhánh trong trường hợp này, không phải lúc nào cũng vậy)

Vì L= 5, mỗi Node lá có từ 3 tới 5 dữ liệu. Điều kiện mọi Node lá phải tối thiểu L/2 đảm bảo B-tree không phải cây nhị phân đơn giản.



Có rất nhiều cách triển khai B-tree khác so với cấu trúc ở trên, nhưng nhìn chung thì giống nhau.

Mỗi Node đại diện chó một block của đĩa cứng, ta chọn số M và L dựa trên cơ sở kích thước dữ liệu.

Ví dụ 1 block đĩa cứng = 8,192 bytes. Xem lại ví dụ về dữ liệu lái xe bang Florida:

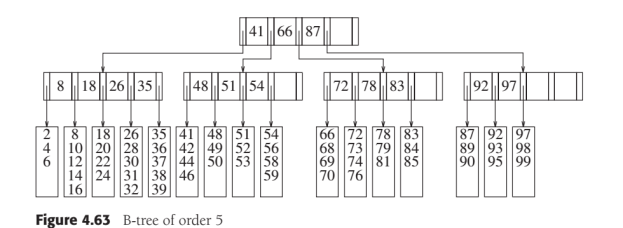
* Cây B-M nhánh có M-1 khóa, với khóa 32-bytes, tổng cộng là 32\*M – 32 bytes khóa
* Mỗi nhánh(Node không phải lá) thực ra là số block khác của đĩa, giả sử 1 nhánh là 4-bytes, tổng cộng 4\*M bytes
* Vùng nhớ cần thiết cho các Node không phải lá là 36\*M – 32 bytes
* Giá trị lớn nhất của M để vùng nhớ không lớn hơn 8.192 bytes là 228. Do đó mỗi Node lá chứa tối đa = 8,192 / 32-byte = 256 dữ liệu, có thể nhét vừa 32 records / 1 block, ta chọn L = 32.

Chúng ta đảm bảo mỗi Node Lá có từ 16 – 32 dữ liệu, có ít nhất 114 Node không lá (trừ root).

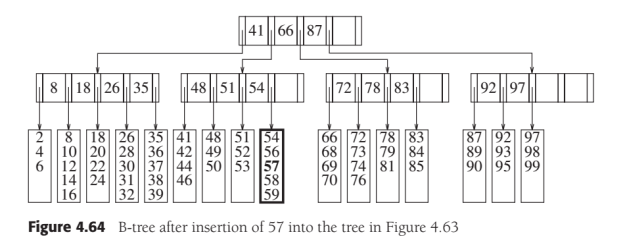
Cho 10 triệu dữ liệu, sẽ có ít nhất 625.000 lá.

Trường hợp xấu nhất cây có 4 bậc, số lần truy xuất xấp xỉ logM/2(N). ( Có thể giảm còn 1, vì root và bậc kế tiếp có thể lưu trong vùng nhớ chính, truy xuất đĩa chỉ cần cho bậc thứ 3 trở đi).

1. Chèn

Phần khó khăn nữa là chèn và xóa dữ liệu trên cây B. Giả sử chúng ta muốn chèn 57 vào cây B- 5 nhánh , có L = 5 

Việc duyệt trên xuống chỉ ra 57 chưa có, chúng ta có thể chèn vào Node lá 54, còn trống 1 vị trí.



1. Phương pháp chia Node

Phương pháp chia Node phải tuân theo 2 đặc tính cây B:

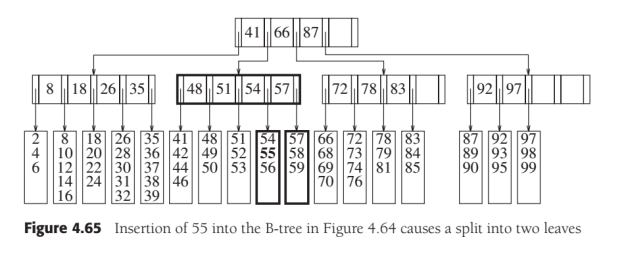
1. Tất cả Node không phải lá (ngoại trừ root) có từ M/2 tới M nhánh con
2. Tất cả Node lá có cùng độ sâu và có từ L/2 tới L dữ liệu.

Cả 2 Node phải thỏa dữ liệu tối thiểu L/2 ( 5 /2 ) , 3 dữ liệu mỗi Node.

Thao tác chia Node cần 2 lần ghi đĩa cho 2 Node mới và lần truy xuất thứ 3 để cập nhật lại Node cha.

Tiếp tục chèn 55, tình huống bây giờ Node Lá 54 đã đầy. Giải pháp là chia thành 2 Node lá.

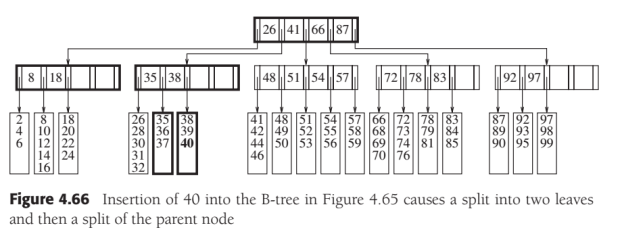
Tại Node cha, cần điều chỉnh lại nhánh con và khóa, kết quả như hình bên dưới.



Việc chia Node hiếm khi xảy ra. Nếu L = 32, khi xảy ra chia Node, 2 Node với 16 và 17 được tạo.

Trường hợp trên Node cha còn trống 1 nhánh nên có thể chia. Nếu Node cha đầy thì sao? Khi ta chèn 40 vào cây, phải chia Node lá 35 thành 2 Node.

Nhưng Node lá đã đầy và Node cha cũng đầy, giải pháp là lại phải chia Node cấp cao hơn như hình bên dưới:



Thao tác này chỉ tốn thêm 2 lần ghi đĩa (5 lần tổng cộng cho việc chèn)

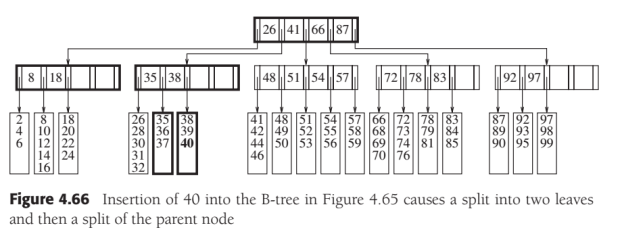
1. Giải thuật:

Khi chia Node không lá, Node cha của nó sẽ thêm 1 nhánh mới.

* Nếu Node lá chưa đầy: chèn dữ liệu vào.
* Nếu Node lá đầy : Xét Node cha cao hơn còn trống và chia
* Nếu đi tới Node root: ta bắt buộc phải chia thành 2 Node root.

Chia Node root là trường hợp duy nhất khiến cây B tăng chiều cao, điều này rất hiếm khi xảy ra như việc chia Node không lá vậy.

Có một kỹ thuật kiểm soát việc quá tải Node con:



Ví dụ khi chèn 29 vào cây B trên, nó đi tới nhánh phải 26 -> nhánh trái 35, Node lá đã đầy.

Chúng ta tạo chỗ trống bằng cách chuyển 32 vào Node lá kế tiếp bên phải khóa 35 của Node cha, đồng thời điều chỉnh giá trị khóa 35 -> 32.

Dù vậy, kỹ thuật này không tạo thêm khoảng trống nào cho Node lá cả.

1. Xóa

We can perform deletion by finding the item that needs to be removed and then remov

ing it. The problem is that if the leaf it was in had the minimum number of data items, then

it is now below the minimum. We can rectify this situation by adopting a neighboring item,

if the neighbor is not itself at its minimum. If it is, then we can combine with the neighbor

to form a full leaf. Unfortunately, this means that the parent has lost a child. If this causes

the parent to fall below its minimum, then it follows the same strategy. This process could

percolate all the way up to the root. The root cannot have just one child (and even if this

were allowed, it would be silly). If a root is left with one child as a result of the adoption

process, then we remove the root and make its child the new root of the tree. This is the

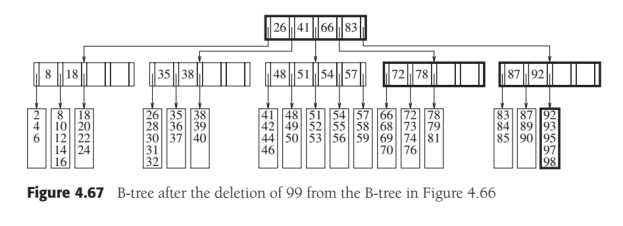
only way for a B-tree to lose height. For example, suppose we want to remove 99 from

the B-tree in Figure 4.66. Since the leaf has only two items and its neighbor is already at

its minimum of three, we combine the items into a new leaf of five items. As a result, the

parent has only two children. However, it can adopt from a neighbor, because the neighbor

has four children. As a result, both have three children. The result is shown in Figure 4.67.



##### Set và map trong STL

STL hỗ trợ thêm 2 container hỗ trợ tìm kiếm nhanh hơn so với vector và list: set và map.

Bảo đảm chi phí các thao tác chèn, xóa ,tìm kiếm theo logN.

1. Sets

Set là cotainer có thứ tự, không cho phép đúp phần tử, cách truy xuất trên set cũng tương tự như vector, list.

Kiểu lồng iterator và const\_iterator cho phép duyệt set, nhiều phương thức triển khai trên vector, list đều có trên set:

* Begin
* End
* Size
* Empty

Phương thức print template cũng sẽ làm việc với set:

template <typename Container>

void print( const Container & c, ostream & out = cout ) {

if( c.empty( ) )

out << "(empty)";

else

{

auto itr = begin( c ); // itr is a Container::const\_iterator

out << "[ " << \*itr++; // Print first item

while( itr != end( c ) )

out << ", " << \*itr++;

out << " ]" << endl;

}

}

Các phương thức riêng triển khai trên set là : insert, remove và search cơ bản.

Vì set không cho phép đúp phần tử, nên có khả năng insert sẽ thất bại. Insert sẽ trả về iterator đại diện cho phần tử mới chèn, hoặc phần tử đã tồn tại trên set khi thất bại.

Cách trả về iterator khá tiện lợi khi cần xóa, vì chúng ta đã biết được vị trí phần tử, tránh thao tác tìm kiếm không cần thiết.

1. Pair

STL còn định nghĩa thêm lớp template **pair**, gồm 2 thành viên struct để truy xuất 2 phần tử theo cặp. Có 2 cách chèn:

* **pair**<iterator,bool> insert( const **Object** & x ): Hàm insert 1 tham số đã mô tả ở trên.
* **pair**<iterator,bool> insert( **iterator** hint, const **Object** & x ): Hàm insert 2 tham số định nghĩa thêm **hint**: ám chỉ vị trí phần tử chèn x nên tới

Nếu chỉ dẫn **hint** chính xác, thao tác chèn sẽ rất nhanh với O(1). Ngược lại, insert sẽ dùng thuật toán của nó.

Ví dụ: đoạn code sau chạy rất nhanh dùng insert 2 tham số

#include <set>

void initSet(){

set<int> s;

for( int i = 0; i < 1000000; ++i )

s.insert( s.end( ), i );

}

Có nhiều phiên bản erase:

* **int** erase( const **Object** & x ): xóa x nếu tìm thấy và trả về số phần tử đã xóa ( 0 hoặc 1)
* **iterator** erase( **iterator** itr ): Xóa theo vị trí iterator cung cấp, trả về iterator của phần tử kế tiếp, itr trở nên vô giá trị.
* **iterator** erase( **iterator** start, **iterator** end ): Xóa tất cả phần tử có iterator bắt đầu từ start , nhưng không gồm end.

Set không dùng contain() mà triển khai find() , trả về iterator đại diện của phần tử (hoặc iterator end nếu không tìm thấy).

* **iterator** find( const **Object** & x ) const;

Theo mặc định, việc sắp xếp dùng đối tượng hàm less<Object>, triển khai gọi toán tử bé hơn (<) của tham số Object.

Ngoài ra khi khởi tạo set, chúng ta có thể triển khai đối tượng hàm so sánh cho tham số thứ hai của set.

Ví dụ: một set string, không kiểm tra hoa thường dùng đối tượng hàm CaseInsensitiveCompare

set<string,CaseInsensitiveCompare> s;

s.insert( "Hello" ); s.insert( "HeLLo" );

cout << "The size is: " << s.size( ) << endl;

1. Maps

Kiểu map dùng chứa các phần tử chỉ gồm khóa và giá trị: <keyType, valueType>.

Key phải là duy nhất, nhưng nhiều key có thể ánh xạ tới cùng 1 giá trị.

Key được duy trì theo trật tự sắp xếp nhất định.

Hành vi của map giống với set, tất cả hàm so sánh xoay quanh key. Map hỗ trợ :

* begin, end, size và empty
* các phương thức insert, find và erase

Iterator là một cặp key-value.

Phương thức insert() nhận đối tượng pair<keyType,valueType> .

Phương thức find() nhận tham số key, trả về iterator, nếu không tìm thấy trả về iterator end()

Các phương thức trên hơi tốn thời gian, may mắn là ta có toán tử truy xuất ngẫu nhiên [ ] được quá tải cho map:

**ValueType** & operator[] ( const **KeyType** & key );

* Nếu key tồn tại trên map, tham chiếu tới giá trị được trả về.
* Nếu key không tìm thấy, nó được chèn với giá trị mặc định và trả về tham chiếu giá trị.

Giá trị mặc định ở đây sẽ là tham số zero cho hàm dựng hay zero với kiểu cơ bản.

Do quan điểm triển khai như trên nên map không có accessor cho toán tử [ ], ví dụ khi truyền tham số là const reference cho map, toán tử [ ] sẽ không dùng được.

1. Truy xuất set- map

Có hai kỹ thuật truy xuất trong map:

* Dùng toán tử [ ] chèn và gán giá trị
* Dùng iterator

map<string,double> salaries;

salaries[ "Pat" ] = 75000.00; //Chèn Pat, gán giá trị 75000

cout << salaries[ "Pat" ] << endl;

cout << salaries[ "Jan" ] << endl; //Chèn Jan, gán giá trị mặc định 0.0

map<string,double>::const\_iterator itr;

itr = salaries.find( "Chris" ); //Dùng find lấy iterator tương ứng với key = “Chris”

if( itr == salaries.end( ) )

cout << "Not an employee of this company!" << endl;

else

cout << itr->second << endl;

Nếu không tìm thấy itr sẽ trỏ tới iterator end(). Nếu thấy, chúng ta có thể dùng itr->second để thay đổi giá trị.

Các phương thức cơ bản insert, erase và find được đảm bảo hiệu suất logN trong trường hợp xấu nhất, do đó cây nhị phân cân bằng được triển khai, nhưng AVL tree không được dùng, mà là top-dow red-black tree.

1. Kỹ thuật triển khai Iterator

Điểm quan trọng là hỗ trợ iterator class, vẫn duy trì con trỏ \*current. Phần khó là đi tới node kế tiếp, có vài giải pháp như sau:

1. Mỗi iterator đã khởi tạo chứa mảng item phải duyệt: không khả thi vì quá tốn chi phí duy trì và phức tạp
2. Iterator duy trì một stack Node trên đường tới current node: iterator trở nên lớn và phức tạp.
3. Mỗi node duy trì thêm liên kết node cha: dù iterator không lớn nhưng vẫn phức tạp
4. Mỗi node duy trì thêm 2 liên kết: 1 tới node nhỏ hơn và 1 tới node lớn hơn, tốn vùng nhớ nhưng code lại đơn giản
5. Duy thì thêm 1 liên kết cho node có liên kết trái hay phải là nullptr, bằng cách dùng biến Boolean để thông báo liên kết trái là thuộc cây nhị phân tìm kiếm chuẩn hay liên kết tới node nhỏ hơn, tương tự cho liên kết phải. Ý tưởng này gọi là threaded tree và dùng trong nhiều triển khai STL.
6. Ứng dụng

Many words are similar to other words. For instance, by changing the first letter, the word

wine can become dine, fine, line, mine, nine, pine, or vine. By changing the third letter, wine

can become wide, wife, wipe, or wire, among others. By changing the fourth letter, wine can

become wind, wing, wink, or wins, among others. This gives 15 different words that can

be obtained by changing only one letter in wine. In fact, there are over 20 different words,

some more obscure. We would like to write a program to find all words that can be changed

into at least 15 other words by a single one-character substitution. We assume that we have

a dictionary consisting of approximately 89,000 different words of varying lengths. Most

words are between 6 and 11 characters. The distribution includes 8,205 six-letter words,

11,989 seven-letter words, 13,672 eight-letter words, 13,014 nine-letter words, 11,297

ten-letter words, and 8,617 eleven-letter words. (In reality, the most changeable words are

three-, four-, and five-letter words, but the longer words are the time-consuming ones to

check.)

The most straightforward strategy is to use a map in which the keys are words and

the values are vectors containing the words that can be changed from the key with a

one-character substitution. The routine in Figure 4.69 shows how the map that is even

tually produced (we have yet to write code for that part) can be used to print the required

answers. The code uses a range for loop to step through the map and views entries that are

pairs consisting of a word and a vector of words. The constant references at lines 4 and 6

are used to replace complicated expressions and avoid making unneeded copies.

The main issue is how to construct the map from an array that contains the 89,000

words. The routine in Figure 4.70 is a straightforward function to test if two words are

identical except for a one-character substitution. We can use the routine to provide the

simplest algorithm for the map construction, which is a brute-force test of all pairs of words.

This algorithm is shown in Figure 4.71.

If we find a pair of words that differ in only one character, we can update the map at

lines 12 and 13. The idiom we are using at line 12 is that adjWords[str] represents the

vector of words that are identical to str, except for one character. If we have previously

seen str, then it is in the map, and we need only add the new word to the vector in the

map, and we do this by calling push\_back. If we have never seen str before, then the act of

using operator[] places it in the map, with a vector of size 0, and returns this vector, so the

push\_back updates the vector to be size 1. All in all, a super-slick idiom for maintaining a

map in which the value is a collection.

void printHighChangeables( const map<string,vector<string>> & adjacentWords,

int minWords = 15 )

{

for( auto & entry : adjacentWords )

{

const vector<string> & words = entry.second;

if( words.size( ) >= minWords )

{

cout << entry.first << " (" << words.size( ) << "):";

for( auto & str : words )

cout << " " << str;

cout << endl;

}

}

}

Given a map containing words as keys and a vector of words that differ in

only one character as values, output words that have minWords or more words obtainable

by a one-character substitution

The problem with this algorithm is that it is slow and takes 97 seconds on our com

puter. An obvious improvement is to avoid comparing words of different lengths. We can

do this by grouping words by their length, and then running the previous algorithm on

each of the separate groups.

To do this, we can use a second map! Here the key is an integer representing a word

length, and the value is a collection of all the words of that length. We can use a vector to

store each collection, and the same idiom applies. The code is shown in Figure 4.72. Line

8 shows the declaration for the second map, lines 11 and 12 populate the map, and then an

extra loop is used to iterate over each group of words. Compared to the first algorithm,

the second algorithm is only marginally more difficult to code and runs in 18 seconds, or

about six times as fast.

// Returns true if word1 and word2 are the same length

// and differ in only one character.

bool oneCharOff( const string & word1, const string & word2 )

{

if( word1.length( ) != word2.length( ) )

return false;

int diffs = 0;

for( int i = 0; i < word1.length( ); ++i )

if( word1[ i ] != word2[ i ] )

if( ++diffs > 1 )

return false;

return diffs == 1;

}

Routine to check if two words differ in only one character

Computes a map in which the keys are words and values are vectors of words

that differ in only one character from the corresponding key.

Uses a quadratic algorithm.

map<string,vector<string>> computeAdjacentWords( const vector<string> & words )

{

map<string,vector<string>> adjWords;

for( int i = 0; i < words.size( ); ++i )

for( int j = i + 1; j < words.size( ); ++j )

if( oneCharOff( words[ i ], words[ j ] ) )

{

adjWords[ words[ i ] ].push\_back( words[ j ] );

adjWords[ words[ j ] ].push\_back( words[ i ] );

}

return adjWords;

}

Function to compute a map containing words as keys and a vector of words

that differ in only one character as values. This version runs in 1.5 minutes on an 89,000-

word dictionary.

Our third algorithm is more complex and uses additional maps! As before, we group the

words by word length, and then work on each group separately. To see how this algorithm

works, suppose we are working on words of length 4. First we want to find word pairs,

such as wine and nine, that are identical except for the first letter. One way to do this is

as follows: For each word of length 4, remove the first character, leaving a three-character

word representative. Form a map in which the key is the representative, and the value is

a vector of all words that have that representative. For instance, in considering the first

character of the four-letter word group, representative "ine" corresponds to "dine", "fine",

"wine", "nine", "mine", "vine", "pine", "line". Representative "oot" corresponds to "boot",

"foot", "hoot", "loot", "soot", "zoot". Each individual vector that is a value in this latest

map forms a clique of words in which any word can be changed to any other word by a

one-character substitution, so after this latest map is constructed, it is easy to traverse it

and add entries to the original map that is being computed. We would then proceed to

the second character of the four-letter word group, with a new map, and then the third

character, and finally the fourth character.

The general outline is

for each group g, containing words of length len

for each position p (ranging from 0 to len-1)

{

Make an empty map<string,vector<string>> repsToWords

for each word w

{

Obtain w’s representative by removing position p

Update repsToWords

}

Use cliques in repsToWords to update adjWords map

}

contains an implementation of this algorithm. The running time improves

to two seconds. It is interesting to note that although the use of the additional maps makes

the algorithm faster, and the syntax is relatively clean, the code makes no use of the fact

that the keys of the map are maintained in sorted order.

// Computes a map in which the keys are words and values are vectors of words

// that differ in only one character from the corresponding key.

// Uses an efficient algorithm that is O(N log N) with a map

map<string,vector<string>> computeAdjacentWords( const vector<string> & words )

{

map<string,vector<string>> adjWords;

map<int,vector<string>> wordsByLength;

// Group the words by their length

for( auto & str : words )

wordsByLength[ str.length( ) ].push\_back( str );

// Work on each group separately

for( auto & entry : wordsByLength )

{

const vector<string> & groupsWords = entry.second;

int groupNum = entry.first;

// Work on each position in each group

for( int i = 0; i < groupNum; ++i )

{

// Remove one character in specified position, computing representative.

// Words with same representatives are adjacent; so populate a map ...

map<string,vector<string>> repToWord;

for( auto & str : groupsWords )

{

string rep = str;

rep.erase( i, 1 );

repToWord[ rep ].push\_back( str );

}

// and then look for map values with more than one string

for( auto & entry : repToWord )

{

const vector<string> & clique = entry.second;

if( clique.size( ) >= 2 )

for( int p = 0; p < clique.size( ); ++p )

for( int q = p + 1; q < clique.size( ); ++q )

{

adjWords[ clique[ p ] ].push\_back( clique[ q ] );

adjWords[ clique[ q ] ].push\_back( clique[ p ] );

}

}

}

}

return adjWords;

}

Function to compute a map containing words as keys and a vector of words

that differ in only one character as values. This version runs in 2 seconds on an 89,000-

word dictionary

As such, it is possible that a data structure that supports the map operations but does

not guarantee sorted order can perform better, since it is being asked to do less. Chapter 5

explores this possibility and discusses the ideas behind the alternative map implementation

that C++11 adds to the Standard Library, known as an unordered\_map. An unordered map

reduces the running time of the implementation from 2 sec to 1.5 sec.

1. Tổng kết

We have seen uses of trees in operating systems, compiler design, and searching.

Expression trees are a small example of a more general structure known as a parse tree,

which is a central data structure in compiler design. Parse trees are not binary, but are

relatively simple extensions of expression trees (although the algorithms to build them are

not quite so simple).

Search trees are of great importance in algorithm design. They support almost all the

useful operations, and the logarithmic average cost is very small. Nonrecursive implemen

tations of search trees are somewhat faster, but the recursive versions are sleeker, more

elegant, and easier to understand and debug. The problem with search trees is that their

performance depends heavily on the input being random. If this is not the case, the run

ning time increases significantly, to the point where search trees become expensive linked

lists.

We saw several ways to deal with this problem. AVL trees work by insisting that all

nodes’ left and right subtrees differ in heights by at most one. This ensures that the tree

cannot get too deep. The operations that do not change the tree, as insertion does, can

all use the standard binary search tree code. Operations that change the tree must restore

the tree. This can be somewhat complicated, especially in the case of deletion. We showed

how to restore the tree after insertions in O(log N) time.

We also examined the splay tree. Nodes in splay trees can get arbitrarily deep, but after

every access the tree is adjusted in a somewhat mysterious manner. The net effect is that

any sequence of M operations takes O(M log N) time, which is the same as a balanced tree

would take.

B-trees are balanced M-way (as opposed to 2-way or binary) trees, which are well

suited for disks; a special case is the 2–3 tree (M = 3), which is another way to implement

balanced search trees.

In practice, the running time of all the balanced-tree schemes, while slightly faster

for searching, is worse (by a constant factor) for insertions and deletions than the simple

binary search tree, but this is generally acceptable in view of the protection being given

against easily obtained worst-case input. Chapter 12 discusses some additional search tree

data structures and provides detailed implementations.

A final note: By inserting elements into a search tree and then performing an inorder

traversal, we obtain the elements in sorted order. This gives an O(N log N) algorithm to

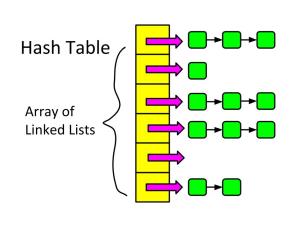
sort, which is a worst-case bound if any sophisticated search tree is used. We shall see

better ways in Chapter 7, but none that have a lower time bound.

### Bảng băm

Nếu bạn làm việc trên C++, bạn có thể dùng std::map trong thư viện STL cho để triển khai mảng khóa dựa trên cấu trúc cây nhị phân.

C++11 hỗ trợ thêm 2 container là std::unordered\_map và std::unordered\_set dựa trên cấu trúc bảng băm.

Bảng băm là sự kết hợp khả năng truy xuất ngẫu nhiên của mảng với tính linh hoạt của danh sách liên kết. Giúp chúng ta hạn chế điểm yếu : tính cố định của mảng (không co giãn được) và tìm kiếm chậm của DSLK.

Tìm kiếm trên DSLK lại kém hiệu quả, không thể truy xuất trực tiếp node, thay vào đó bạn phải duyệt danh sách cho tới khi tìm được, trung bình mất O(n/2).

Tìm kiếm, xóa, chèn trong bảng băm mất thời gian trung bình.

Bảng băm không phải cấu trúc tốt cho việc sắp xếp dữ liệu, nên không có các thao tác như findMin, findMax.

##### Mảng khóa và mảng index

Mảng trong C hay còn gọi là index array, dùng chỉ số index để truy xuất phần tử ngẫu nhiên . Ví dụ:

employees[50];

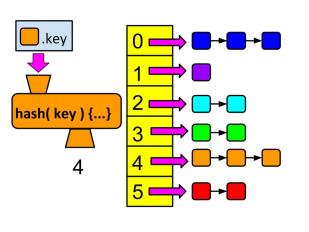
Trong mảng khóa, kết hợp với mỗi khóa sẽ là thông tin bất kì, như số model, tên.. Nên nếu bạn có mảng khóa với thông tin nhân viên, bạn có thể truy cập nhân viên “John Brown” như sau:

employees["Brown, John"];

Một dạng cơ bản của mảng khóa, gọi là bảng băm. Trong bảng băm, khóa được dùng để kiếm phần tử thay vì chỉ số index. Vì bảng băm phải được triển khai dùng mảng index, nên sẽ có vài phương thức chuyển khóa sang số index, gọi là hàm băm

##### Hàm băm

Hàm băm quyết định cách lưu trữ và lấy dữ liệu trong bảng băm. Nó nhận tham số khóa và trả về chỉ số index vị trí phần tử.

Ví dụ: hàm băm hash() nhận tham số khóa kiểu string, nó tính tổng giá trị ASCII các kí tự.

Sau đó trả lấy số dư giá trị này với kích thước bảng băm:

#define TABLE\_LENGHT 100

int hash( string key )

{

int value = 0;

for ( int i = 0; i < key.length(); i++ )

value += key[i];

return value % TABLE\_LENGHT;

}

Một điều quan trọng là hàm băm phải trả về cùng giá trị mọi lần nó nhận được cùng một khóa.

##### Chọn thuật toán băm tốt

Trên thực tế, hàm băm có thể gây ra hiện tượng xung đột khi ánh xạ hai hay nhiều khóa vào cùng một vị trí trên bảng băm.

Cho dù số phần tử của bạn có ít, thì cách duy nhất để tránh xung đột là kích thước bảng băm phải đủ lớn cho mỗi phần tử là một ô trống.

Việc mở rộng bảng băm sẽ tốn nhiều bộ nhớ . Vì thế, hàm băm phải có tính hiệu quả cao.

Hàm băm cần 2 đặc tính sau:

* Đơn giản, tính toán nhanh
* Phân phối khóa đều : cách phổ biến nhất luôn là dùng phép chia lấy dư cho kích thước bảng băm, để giới hạn giá trị băm luôn nằm trong khoảng kích thước bảng.

Thuật toán hàm băm lệ thuộc vào nhiều yếu tố :

* Khóa là kiểu số hay chuỗi: mỗi kiểu sẽ có các cách băm hiệu quả khác nhau.
* Đặc tính của khóa: có tính ngẫu nhiên hay lặp lại theo thói quen. Nếu khóa lặp lại nhiều, hàm băm sẽ cho ra các giá trị giống nhau, dẫn tới xung đột.
* Kích thước bảng băm: nếu bạn triển khai bảng băm có kích thước lớn thì thuật toán cần tính toán để có thể mở rộng giá trị ra.

Phần tiếp theo điểm qua vài thuật băm đơn giản với kiểu số và chuỗi.

1. Khóa Kiểu Integer
2. Phương Pháp Chọn Số

Nếu khóa của bạn có 9 kí tự số như ID nhân viên 001364825, bạn có thể chọn số tứ 4 và số cuối

h(001364825) = 35 (chỉ số vị trí là 35 % table\_size)

Phương pháp này nhanh, đơn giản nhưng chúng ta sẽ không chọn cách này. Vì nếu khóa không có tính ngẫu nhiên, khả năng xung đột rất cao.

1. Phương Pháp Cộng Dồn

Bạn có thể dùng cho chuỗi số hay kí tự đều được

Cho chuỗi số 001364825:

0 + 0 + 1 + 3 + 6 + 4 + 8 + 2 + 5 = 29 (chỉ số vị trí là 29 % table\_size)

Bạn để ý, chuỗi 9 số sẽ có giá trị tối đa là: 9\*9 = 81. Một kỹ thuật được dùng để tăng giá trị lên cho phù hợp với kích thước bảng băm lớn là nhóm các kí tự số thành số lớn hơn rồi cộng dồn:

Ví dụ ta nhóm từng 3 số:

001 + 364 + 825 = 1,190

Giá trị tối đa là 3 \* 999 = 2,997

Tùy theo kích thước bảng băm mà bạn nhóm nhiều hay ít số lại. Ngoài ra bạn có thể kết hợp phương pháp chọn số và phương pháp nhóm số rồi cộng dồn lại

1. Chọn Kích Thước Bảng Là Số Nguyên Tố

Phép lấy dư luôn hiệu quả trong việc giới hạn giá trị trong khoảng cho phép và phân phối khóa đều trên bảng băm.

h(x) = x % table\_size

Nếu kích thước bảng là số chẵn hay số tròn như 100, 200, 300, 400, 500, 600 … bội số của 10 đều nằm tại vị trí zero, dẫn tới xung đột, phân phối khóa không đều.

Cách tốt nhất là chọn kích thước bảng bằng số nguyên tố như 101, vừa tránh xung đột , vừa giúp phân phối tốt hơn.

1. Khóa Là Kiểu Chuỗi
2. Phương pháp cộng dồn:

Một trong các giải pháp phổ biến là cộng dồn giá trị ASCII mỗi kí tự của khóa, sau đó lấy dư phép chia với kích thước bảng.

//thuật băm khá phổ biến

int hash\_function(const string &key) {

int index = 0;

for (char c : key) {

index += c;

}

return index % size;

}

Thuật toán trên khá nhanh, ta đảm bảo kích thước bảng là số nguyên tố.

Xét khóa là chuỗi từ 8 kí tự trở xuống.

Hầu như mọi kí tự ASCII đều có giá trị lớn nhất là 127. Mỗi kí tự trong khóa sẽ có giá trị tối đa 127, tức 127\*8 = 1016. Hàm băm sẽ cho ra giá trị từ 0 tới 1016. Nên chọn kích thước là 1017

Nếu bạn đang làm việc với khóa là chuỗi tên người dùng (Tiếng Anh), bạn nên bỏ qua kí tự đầu – theo thống kê tên bắt đầu với S sẽ nhiều hơn là Z, tỉ lệ không đều cho các chữ cái alphabet.

1. Phương pháp chọn kí tự:

Giả sử ta chọn 3 kí tự đầu tiên để băm. Khóa là tên người gồm ít nhất 3 kí tự.

int hash\_function(const string & key)

{

return ( key[ 0 ] + 27 \* key[ 1 ] + 729 \* key[ 2 ] ) % size;

}

Kí tự alphabet tiếng Anh gồm 26 chữ cái bỏ qua khoảng trắng. Về lý thuyết có 263 = 17.576 khả năng kết hợp của 3 kí tự (bỏ qua khoảng trắng).

Nhưng tên người trong tiếng Anh không mang tính ngẫu nhiên, qua khảo sát từ điển, tổng cộng chỉ có 2.851 khả năng mà thôi.

Do đó, chúng ta chọn kích thước bảng là 2852.

1. Định luật horner:

/\*thuật băm tốt hơn.\*/

unsigned int hash(const string & key)

{

unsigned int hashVal = 0;

for( char c : key )

hashVal = 37 \* hashVal + c;

return hashVal % size;

}

Với khóa gồm n kí tự, ta có công thức tổng sigma dãy số của vòng lặp là :

Đoạn code tính đa thức f(37) theo định luật Horner. Ví dụ một cách khác để tính

Là

Định luật Horner mở rộng cho tới đa thức bậc n.

Hàm băm trên có ưu điểm là cho phép tràn số (hashVal có kiểu số không âm). Nó đơn giản tối đa tính toán và tốc độ nhanh.

Nếu khóa quá dài, hàm băm mất nhiều thời gian hơn để tính. Thực tế khi triển khai, chúng ta không dùng mọi kí tự của khóa.

Độ dài và thuộc tính khóa mới ảnh hưởng lên thuật toán. Ví dụ: ta có địa chỉ gồm tên đường. Hàm băm sẽ lấy cặp kí tự từ tên đường và một cặp kí tự từ tên thành phố , Zip code.

Một số lập trình viên chỉ dùng kí tự theo thứ tự lẻ, với ý tưởng tiết kiệm thời gian tính toán bù cho khả năng phân phối khóa hơi yếu một chút.

Vấn đề còn lại là giải quyết xung đột. Chúng ta sẽ nói về hai phương pháp chính là : chuỗi liên kết và đánh địa chỉ mở

##### Dùng DSLK

1. Xung đột

Tình huống ngẫu nhiên, một số từ khác nhau cho cùng kết quả qua hàm băm. Ví dụ khóa “tar” và “rat” đều hash ra 2 nếu bảng có kích thước 13. Chúng ta có xung đột

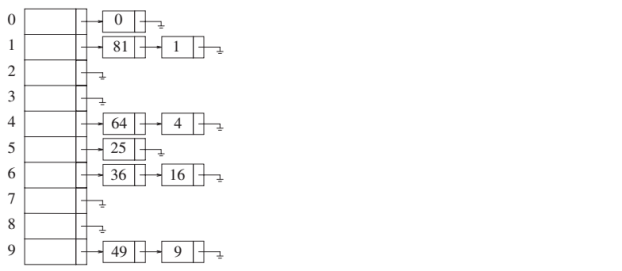
1. Xử lí xung đột với chuỗi liên kết

Chiến thuật là tạo một chuỗi danh sách liên kết các phần tử cùng giá trị băm.

Để tối ưu khả năng lưu trữ và lấy dữ liệu, chúng ta phải tối thiểu xung đột. Mỗi xung đột xảy ra trên dữ liệu trong bucket, một node trên DSLK. Chúng ta biết rằng DSLK kém hiệu quả trong việc lưu trữ và lấy dữ liệu khi trở nên lớn hơn, nên phải giữ cho DSLK càng ngắn càng tốt.

Dĩ nhiên, bạn có thể tạo một mảng cực lớn nhằm hạn chế xung đột, nhưng cách đó không đúng với mục đích của bảng băm, đó là ưu thế của tốc độ và nhỏ gọn.

Để cho tiện, chúng ta chọn kích thước bảng là 10, hàm băm đơn giản là hash(x) = x mod 10. Bạn xem kết quả hình minh họa bên dưới



Hình 10: Bảng băm chuỗi rời rạc

Để tìm kiếm, bạn dùng hàm băm kiếm vị trí node đầu tiên của danh sách, sau đó duyệt tới cuối.

Khi chèn, để ý trường hợp trùng dữ liệu và khóa, tùy theo mục đích sử dụng mà bạn có thể ghi đè, tạo mới hay bỏ qua. Thường chúng ta chèn vào đầu danh sách, vì nó tiện.

1. Triển khai

Khi triển khai, bạn có thể dùng thư viện STL. Nếu vùng nhớ khan hiếm, thì bạn không nên dùng chúng (vì các bộ chứa STL là kiểu DSLK kép và tốn bộ nhớ).

Khai báo lớp bảng băm:

template <typename HashedObj>

class HashTable

{

public:

explicit HashTable( int size = 101 );

bool contains( const HashedObj & x ) const;

void makeEmpty( );

bool insert( const HashedObj & x );

bool insert( HashedObj && x );

bool remove( const HashedObj & x );

private:

vector<list<HashedObj>> theLists; // mảng vector<> gồm các phần tử kiểu danh sách List<>

int currentSize;

void rehash( );

size\_t myhash( const HashedObj & x ) const;

};

Lớp bảng băm chứa mảng danh sách liên kết. Mảng cấp phát trong hàm dựng.

Cú pháp khởi tạo theLists theo chuẩn C++11, cũng như cây nhị phần tìm kiếm triển khai cho đối tượng có tính so sánh, đối tượng **HashedObj** trên bảng băm cũng triển khai hàm băm và toán tử so sánh (== hay !=, hay cả hai).

Lưu ý khi triển khai: Nên tận dụng từ khóa “**auto**” khi duyệt hay thao tác trên list<>, vì chúng ta đang dùng template nên không biết trước kiểu dữ liệu trên lớp chứa list<>.

Triển khai thuật băm bên ngoài dùng template:

template <typename Key>

class hash

{

public:

size\_t operator() ( const Key & k ) const;

};

Lớp bảng băm triển khai thuật toán băm trên mọi kiểu dữ liệu khóa, trả về chỉ số kiểu size\_t.

Mỗi đối tượng HashObj có đặc tính khóa khác nhau, bạn phải chuyên hóa template cho từng kiểu đối tượng.

Ví dụ: chuyên hóa template lớp hash cho kiểu string:

template <>

class hash<string>{

public:

size\_t operator()( const string & key )

{

size\_t hashVal = 0;

for( char ch : key )

hashVal = 37 \* hashVal + ch;

return hashVal;

}

};

Triển khai phương thức băm myhash(), sử dụng giá trị trả về từ đối tượng hàm băm hash<HashedObj>( x )

template <HashedObj>

size\_t HashTable<HashedObj>::myhash( const HashedObj & x ) const

{

static hash<HashedObj> hf;

return hf( x ) % theLists.size( );

}

Triển khai phương thức xóa dữ liệu trên bảng băm: đơn giản là dùng vòng lặp for each và phương thức clear() của list<>

template <HashedObj>

void HashTable<HashedObj>::makeEmpty( )

{

for( auto & thisList : theLists )

thisList.clear( );

}

Triển khai phương thức kiểm tra tồn tại entry trên bảng băm:

* Trước tiên dùng phương thức băm myhash() lấy chỉ số entry trên bảng
* Tiếp theo dùng phương thức find() với iterator để tìm đúng đối tượng
* So sánh iterator trả về từ find() với end() để trả về

**Lưu ý: Bạn phải quá tải phép so sánh “==” khi định nghĩa đối tượng trước khi dùng find().**

template <HashedObj>

bool HashTable<HashedObj>::contains( const HashedObj & x ) const

{

auto & whichList = theLists[ myhash( x ) ];

return find( begin( whichList ), end( whichList ), x ) != end( whichList )

}

Triển khai phương thức xóa entry:

* Dùng phương thức băm để lấy chỉ số index
* Dùng find() tìm đối tượng
* Nếu tìm thấy đối tượng thì xóa, trả về true. Đồng thời giảm số phần tử đi một.
* Ngược lại trả về false.

template <HashedObj>

bool HashTable<HashedObj>::remove( const HashedObj & x )

{

auto & whichList = theLists[ myhash( x ) ];

auto itr = find( begin( whichList ), end( whichList ), x );

if( itr == end( whichList ) )

return false;

whichList.erase( itr );

--currentSize;

return true;

}

Triển khai phương thức chèn entry:

* Tùy theo mục đích mà chúng ta có nhiều cách xử lí. Nguyên tắc chèn là nếu tìm thấy phần tử trên bảng thì trả về false.
* Ngược lại, để thuận tiên, chúng ta sẽ chèn phần tử vào cuối danh sách entry.
* Trường hợp số phần tử vượt qua kích thước qui định của bảng băm, chúng ta sẽ dùng tính năng rehash(), sẽ đề cập ở phần sau.

**Lưu ý: ta dùng biến tham chiếu whichList đại diện cho entry lớp chứa list<> trên bảng**

template <HashedObj>

bool HashTable<HashedObj>::insert( const HashedObj & x )

{

auto & whichList = theLists[ myhash( x ) ];

if( find( begin( whichList ), end( whichList ), x ) != end( whichList ) )

return false;

whichList.push\_back( x );

// Rehash; see Section 5.5

if( ++currentSize > theLists.size( ) )

rehash( );

return true;

}

1. Tối ưu hóa với chỉ số tải

Ngoài danh sách liên kết, bạn có thể triển khai cây nhị phân hay thậm chí một bảng băm trên mỗi entry đều được.

Để danh sách liên kết đơn giản, bảng băm phải đủ lớn, thuật toán băm phân phối tốt.

Chúng ta định nghĩa chỉ số tải λ của bảng băm là tỉ lệ của số phần tử trên kích thước bảng. Ví dụ: số phần tử là 7, kích thước bảng là 10 => λ = 0.7 (chúng ta còn nói chiều dài trung bình của danh sách là 0.7)

Hiệu suất tìm kiếm được đánh giá qua tính phức tạp thuật băm cộng thêm thời gian duyệt danh sách.

* Thời gian tìm kiếm thất bại trung bình sẽ là λ liên kết phải duyệt
* Thời gian tìm kiếm thành công sẽ khoảng 1+ (λ/2) liên kết phải duyệt

Phân tích:

Danh sách duyệt chứa ít nhất là một node là thông tin tìm kiếm. Xét bảng băm gồm N phần tử, có M danh sách liên kết

Số node trung bình phải duyệt (không tính node cần tìm) trên cùng danh sách là: (N-1)/M = λ – 1/M.

M sẽ khá lớn, ta có thể bỏ qua 1/M. Trung bình, một nửa số node khác trên cùng danh sách sẽ phải duyệt để tìm, tức λ/2, cộng thêm node cần tìm sẽ là 1+( λ/2) liên kết phải duyệt.

Phân tích trên chỉ ra rằng kích thước bảng không quan trọng như chỉ số tải λ. Luật chung cho bảng băm chuỗi rời rạc là giữ kích thước bảng băm tương đương với tổng số phần tử (nói cách khác là giữ λ ≈ 1 là tối ưu nhất cho việc tìm kiếm).

Trong phương thức chèn, nếu chỉ số tải bằng 1, chúng ta sẽ mở rộng bảng bằng cách gọi rehash()

##### Không dùng DSLK

Việc sử dụng danh sách liên kết giảm hiệu suất xuống vì chúng cần thời gian để cấp phát entry mới (đặc biệt khi bạn dùng nhiều thứ tiếng khác nhau) và thời gian yêu cầu để triển khai cấu trúc dữ liệu thứ hai ngoài bảng băm.

Một giải pháp khác nhanh hơn là tìm ô trống khác trên bảng.

Giả sử các ô h0(x), h1(x), h2(x), . . . được thử cho việc tìm thay thế, trong đó:

**hi(x) = ( hash(x) + f(i) ) % table\_size // với f(0) = 0**

* Hàm hash(x) : thuật băm sinh ra hash code.
* Hàm f(i): thuật toán giải quyết xung đột.

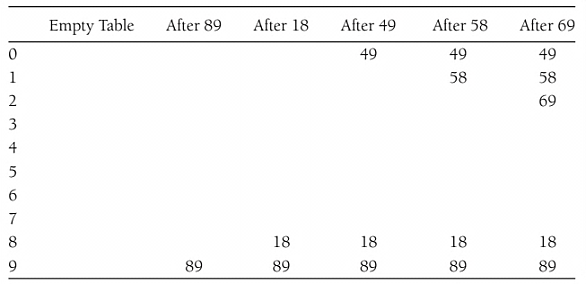
Do không triển khai danh sách liên kết nên tất cả dữ liệu sẽ phải nằm gọn trong bảng băm, việc duy trì bảng băm đủ lớn là cần thiết.

Các bạn nên duy trì chỉ số tải λ <= 0.5. Kiểu bảng băm như vậy chúng ta gọi là bảng băm tuyến tính.

Tiếp theo chúng ta sẽ nói về các chiến thuật để tránh xung đột cho bảng băm kiểu này.

1. Phương pháp tuyến tính

Trong tìm kiếm tuyến tính, hàm f là tuyến tính của i, f(i) = i. Đây là lượng ô liên tiếp cần thử để kiếm ra ô trống.



Bảng băm tìm tuyến tính, sau khi chèn mỗi phần tử

Hình trên là kết quả bảng băm sau khi chèn các khóa {89, 18, 49, 58, 69} , dùng hàm băm và thuật toán tránh xung đột f(i) = i.

* Xung đột đầu tiên xảy ra khi 49 được chèn, thuật toán đặt nó vào điểm trống kế tiếp, điểm zero, đang trống.
* Khóa 58 xung đột với 18, nó thử qua các ô 89, và sau đó là 49 trước khi tìm thấy ô trống.
* Xung đột 69 cũng tương tự. Khi bảng băm vẫn đủ lớn, ô trống sẽ có thể tìm thấy nhanh chóng, nhưng về sau khi bảng càng lớn thời gian tìm cũng lớn hơn.

Như bạn thấy, cụm ô bị chiếm bắt đầu hình thành sau lần chèn khóa đầu tiên. Đây là hiệu ứng đóng cụm, nghĩa là bất kì khóa nào băm vào cụm này qua thuật toán f(i), và nó sẽ được thêm vào cụm.

Mặc dù chúng ta sẽ không làm phép toán, nhưng có thể thấy rằng số bước nhảy cho việc tìm tuyến tính nằm trong khoảng:

Cho việc chèn khóa, hay sau khi tìm không thấy khóa. Và:

Cho việc tìm khóa thành công.

Phân tích từ đoạn code cho thấy khả năng chèn và tìm thất bại có cùng số bước nhảy. khả năng tìm khóa thành công sẽ tốn ít thời gian hơn tìm không thấy khóa.

Công thức tương ứng, nếu việc hình thành cụm không thành vấn đề, khá dễ để rút ra.

Chúng ta giả sử một bảng rất lớn, mỗi bước nhảy độc lập với bước nhảy xảy ra trước đó. Giả thuyết thỏa điều kiện của chiến thuật tránh xung đột ngẫu nhiên và chỉ hợp lý khi λ không gần tới 1.

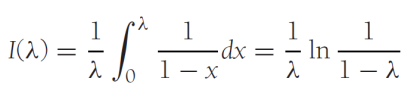
Đầu tiên chúng ta suy luận số bước nhảy trong tìm kiếm không thành công (Đây cũng là số bước cần phải nhảy cho tới khi tìm thấy ô trống)

Vì phân số là mật độ ô trống 1- λ, số ô cần phải tìm cho là 1/(1 − λ). Số ô cần tìm cũng là số ô cần duyệt qua để chèn. Khi một phần tử được chèn, xem như là thao tác tìm khóa không thành công.

Do đó chi phí tìm không thành công được dùng để tính chi phí trung bình tìm thành công.

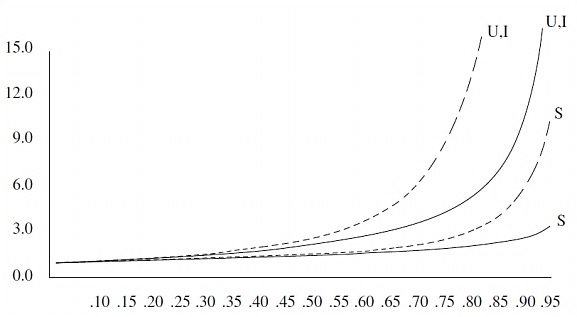
Lưu ý rằng λ thay đổi giá trị từ 0 tới giá trị hiện tại, nên các giá trị khóa chèn sớm tốn ít chi phí hơn và giúp làm giảm chi phí trung bình chèn tất cả khóa.

Như ví dụ trong hình 11, tại bước chèn khóa 18, λ = 0.2. 18 được chèn khi bảng gần như trống, truy xuất dễ hơn so với các phần tử chèn sau. Ước lượng chi phí trung bình dùng tích phân để tính giá trị kỳ vọng của thời gian chèn:



Công thức này rõ ràng khá hơn so với công thức tương ứng cho tìm tuyến tính.

Việc phân cụm không chỉ là vấn đề tồn tại trong học thuyết mà còn trong triển khai.



Number of probes plotted against load factor for linear probing (dashed) and random strategy (S is successful search, U is unsuccessful search, and I is insertion)

Đồ thị trên so sánh hiệu suất của tìm tuyến tính (đường nét đứt) khi giải quyết cá xung đột ngẫu nhiên.

Biến cố tìm kiếm thành công là đường S, và tìm không thành công hay chèn được đánh dấu là U, I tương ứng.

* Nếu λ = 0.75, công thức trên chỉ ra rằng 8.5 là bước nhảy mong đợi cho thao tác chèn tìm tuyến tính.
* Nếu λ = 0.9, thì 50 là bước nhảy mong đợi, đây là mốc không hợp lý.

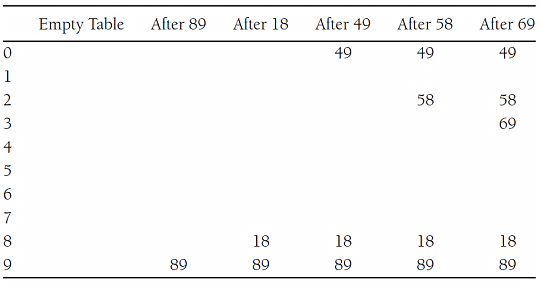
Sự so sánh giữa bước nhảy 4 và 10 cho chỉ số tải tương ứng nếu xảy ra phân cụm không phải vấn đề lớn. Qua công thức này, cho thấy tìm tuyến tính có thể là một ý tưởng tồi nếu bảng băm bị lấp đầy hơn một nửa.

* Nếu λ = 0.5, chỉ 2.5 bước nhảy yêu cầu, tính trung bình cho thao tác chèn, và chỉ 1.5 bước nhảy, tính trung bình cho tìm kiếm thành công

1. Phương pháp bình phương

Giải pháp này nhằm triệt tiêu vấn đề đóng cụm của phương pháp tuyến tính. Phương pháp này dùng hàm bình phương.

Lựa chọn phổ biến là f(i) = i2



Bảng băm tìm theo PT bậc hai sau mỗi lần chèn

Hình trên là kết quả dùng phương pháp bình phương với cùng dữ liệu đầu vào với phương pháp tuyến tính.

* Khi 49 xung đột với 89, vị trí tiếp theo (i2=1) là HashTable[0] đang trống, 49 nhảy vào đây.
* Khóa 58 xung đột tại vị trí HashTable[8], i2=1 xung đột tại HashTable[9], i2=4 với HashTable[2] đang trống, 58 nhảy vào đây.
* Tương tự cho khóa 69 xung đột tại HashTable[9], nó được xếp vào HashTable[3] khi i2=4.

Tìm tuyến tính là ý tưởng khá tồi khi bảng băm gần như đầy, nó triệt tiêu hiệu suất chương trình với O(n) lần duyệt.

Tìm bình phương có phần tồi hơn dù phân phối tốt hơn: không đảm bảo tìm được ô trống khi bảng đầy hơn một nửa, hay thậm chí bảng chưa đầy với kích thước không phải nguyên tố.

Nguyên nhân là hầu như một nửa bảng có thể dùng như vị trí thay thế cho việc xử lí xung đột.

1. Chứng minh định lý

**Định lý:**

**Nếu dùng phương pháp bình phương và kích thước bảng là nguyên tố, phần tử mới luôn tìm được ô trống nếu bảng đang trống ít nhất một nửa.**

**If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.**

Chọn kích thước bảng TableSize là số lẻ nguyên tố lớn hơn 3. Dễ suy ra rằng vị trí thay thế đầu tiên [TableSize/2] (bao gồm vị trí khởi đầu h0(x) ) là rất có khả năng.

We show that the first \_TableSize/2\_ alternative locations (including the initial location h0(x)) are all distinct.

Xét 2 vị trí: h(x) + i2 (mod TableSize) và h(x) + j2 (mod TableSize), với 0 ≤ i, j ≤ [TableSize/2] và i ≠ j

Thì:

h(x) + i2 = h(x) + j2 (mod TableSize)

i2= j2 (mod TableSize)

i2− j2 =0 (mod TableSize)

(i − j)(i + j) =0 (mod TableSize)

Vì TableSize là nguyên tố nên (i − j) hay (i + j) bằng 0 (mod TableSize).

Vì i khác j, mệnh đề đầu tiên là không thể.

Vì 0 ≤ i, j ≤ [TableSize/2] nên mệnh đề thứ 2 là không thể. Do đó, vị trí thay thế [TableSize/2] rất có khả năng.

Khi bảng đầy hơn một nửa, việc chèn sẽ thất bại, điều quan trọng là kích thước phải là nguyên tố, để giảm các bước tìm vị trí thay thế.

Nếu chọn kích thước nguyên tố theo dạng 4k + 3, phương pháp xử lí xung đột bình phương được dùng là: f(i) = ± i2, thì cả bảng có thể duyệt. Chi phí bỏ ra khá lớn vì độ phức tạp.

Ví dụ TableSize = 16, thì chỉ có các khả năng tìm vị trí thay thế cho bước nhảy là 1,4 và 9.

Việc xóa khóa không nên thực hiện trên bảng băm tìm kiếm vì gây xung đột khi đi qua ô này, ví dụ nếu xóa 89, thì các bước tìm kiếm khóa theo sau sẽ thất bại.

Do đó, chỉ có thể xóa tạm thời, mặc dù trường hợp này không ám chỉ như vậy.

Chúng ta sử dụng kiểu enum chuẩn sau:

enum EntryType { ACTIVE, EMPTY, DELETED };

Lớp Interface của bảng băm tìm kiếm có cấu trúc lồng HashEntry:

template <typename HashedObj>

class HashTable{

public:

explicit HashTable( int size = 101 );

bool contains( const HashedObj & x ) const;

void makeEmpty( );

bool insert( const HashedObj & x );

bool insert( HashedObj && x );

bool remove( const HashedObj & x );

enum EntryType { ACTIVE, EMPTY, DELETED };

private:

struct HashEntry{

HashedObj element;

EntryType info;

HashEntry(const HashedObj &e = HashedObj{}, EntryType i:element{e}, info{i} {}

HashEntry(HashedObj &&e, EntryType i = EMPTY):element{std::move(e)}, info{i} {}

};

vector<HashEntry> array;

int currentSize;

bool isActive( int currentPos ) const;

int findPos( const HashedObj & x ) const;

void rehash( );

size\_t myhash( const HashedObj & x ) const;

};

Mỗi ô bảng băm là cấu trúc HashEntry chứa khóa và thành viên “info” mặc định là enum “EMPTY”.

Phương thức “contains(x)” sẽ triệu hồi hai phương thức thành viên private “isActive” và “findPos”.

Phương thức “findPos” dùng để xử lí xung đột.

Chúng ta đảm bảo phương thức chèn “insert” chỉ thực hiện với kích thước bảng lớn hơn ít nhất 2 lần số phần tử đang có, để phương pháp bình phương có thể triển khai được.

We ensure in the insert routine that the hash table is at least twice as large as the number

of elements in the table, so quadratic resolution will always work. In the implementation

in Figure 5.16, elements that are marked as deleted count as being in the table. This can

cause problems, because the table can get too full prematurely. We shall discuss this item

presently.

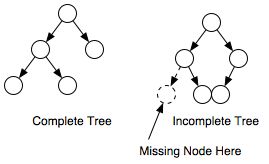
Lines 12 to 15 represent the fast way of doing quadratic resolution. From the definition

of the quadratic resolution function, f(i) = f(i − 1) + 2i − 1, so the next cell to try is a

distance from the previous cell tried and this distance increases by 2 on successive probes.

1. Băm kép

### Heap

Ở một khía cạnh, Heap là một cây nhị phân đã sắp xếp. Mặc dù head hoàn toàn không phải tuân theo thứ tự, nó tuận theo nguyên tắc sắp xếp sau:

* Mọi node đều có giá trị bé hơn (để đơn giản, chúng ta xếp từ nhỏ đến lớn) tất cả cây con của nó.
* Thêm nữa head là cây hoàn chỉnh – tức cây này không có khoảng trống giữa các lá.

Ví dụ: một cây có node gốc, node này chỉ có một một node con gồm node con trái. Theo nguyên tắc cây hoàn chỉnh phải lấp đầy tất cả các cấp trước khi thêm node vào cấp kế tiếp, và một node phải lấp vào node con phải.

##### Tại sao dùng heap?

Heap có thể hiểu là hàng đợi ưu tiên: node quan trọng nhất luôn nằm ở trên cùng, khi xóa, node thay thế phải là quan trọng nhất trong các node còn lại. Điều này giúp ích khi lập trình giải thuật xử lý theo trật tự chắc chắn, nhưng khi bạn không muốn sắp xếp toàn bộ hay cần biết mọi thứ về các node còn lại.

Ví dụ: một giải thuật nổi tiếng về tìm đường đi ngắn nhất giữa các node trong đồ thị, Dijkstra’s, có thể tối ưu hóa dùng hàng đợi ưu tiên.

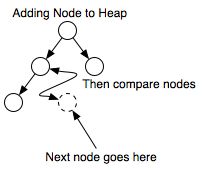
Head còn dùng để sắp xếp dữ liệu. Head sort có độ phức tạp là O(n\*logn), dù vậy nó không phải là giải thuật sắp xếp nhanh nhất.

##### Triển khai head thế nào?

Although the concept of a heap is simple, the actual implementation can appear tricky. How do you remove the root node and still ensure that it is eventually replaced by the correct node? How do you add a new node to a heap and ensure that it is moved into the proper spot?

The answers to these questions are more straight forward than meets the eye, but to understand the process, let's first take a look at two operations that are used for adding and removing nodes from a heap: upheaping and downheaping.

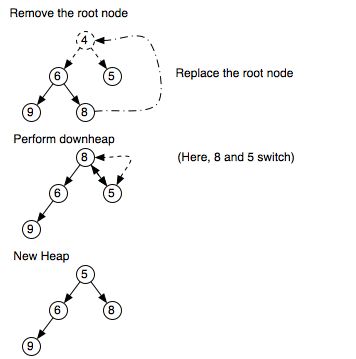
Upheap: The upheap process is used to add a node to a heap. When you upheap a node, you compare its value to its parent node; if its value is less than its parent node, then you switch the two nodes and continue the process. Otherwise the condition is met that the parent node is less than the child node, and so you can stop the process. Once you find a parent node that is less than the node being upheaped, you know that the heap is correct--the node being upheaped is greater than its parent, and its parent is greater than its own parent, all the way up to the root.



Downheap: The downheap process is similar to the upheaping process. When you downheap a node, you compare its value with its two children. If the node is less than both of its children, it remains in place; otherwise, if it is greater than one or both of its children, then you switch it with the child of lowest value, thereby ensuring that of the three nodes being compared, the new parent node is lowest. Of course, you cannot be assured that the node being downheaped is in its proper position -- it may be greater than one or both of its new children; the downheap process must be repeated until the node is less than both of its children.

When you add a new node to a heap, you add it to the rightmost unoccupied leaf on the lowest level. Then you upheap that node until it has reached its proper position. In this way, the heap's order is maintained and the heap remains a complete tree.

Removing the root node from a heap is almost as simple: when you take the node out of the tree, you replace it with "last" node in the tree: the node on the last level and rightmost on that level.



Once the top node has been replaced, you downheap the node that was moved until it reaches its proper position. As usual, the result will be a proper heap, as it will be complete, and even if the node in the last position happens to be the greatest node in the entire heap, it will do no worse than end up back where it started.

##### Độ phức tạp của heap

Whenever you work with a heap, most of the time taken by the algorithm will be in upheaping and downheaping. As it happens, the maximum number of levels of a complete tree is log(n)+1, where n is the number of nodes in the tree. Because upheap or downheap moves an element from one level to another, the order of adding to or removing from a heap is O(logn), as you can make switches only log(n) times, or one less time than the number of levels in the tree (consider that a two level tree can have only one switch).

##### Triển khai

This source code is an implementation of the Heap Tree class and the Heap Sort algorithm. The class is implemented with templates.

For the templated class, the elements must have the operators >, =, and < defined.

To use the Heap sort that is built into the class, two separate steps must be taken. The first is to call the constructor, which organizes the array into a heap:

HeapTree<TYPE> HeapName(Array, Num, MaxNum);

TYPE is the data type of the elements, Array is the actual array to be sorted, and Num is the number of elements that are to be sorted. MaxNum normally sets the limit on the number of data nodes that the Heap can have. If you are only using the heap for sorting, you can set MaxNum to Num. (However, MaxNum should not be set to anything less than Num).

When the constructor is called, the Heap copies the Array. Thus, neither the Array variable nor what it points to will be modified by the Heap.

The second step is to call the actual sort, which will organize the heap into a sorted array:

NewArray \*Sort();

This Sort() function will return a pointer to another array which is sorted. Any modifications done to NewArray or its contents will not affect the heap.

/\*

-------------------------------------------------------------------

| |

| HeapTree Class |

| =========================================================== |

| This HeapTree Class has been implemented with templates. |

| |

| To use the HeapSort that is built into the class, two |

| separate steps must be taken. The first is the constructor: |

| |

| HeapTree<Type> HeapName(Array, Num, MaxNum); |

| |

| 'Type' is the data type of the Array elements, 'Array' is a |

| standard C++ array to be sorted and 'Num' is the number of |

| elements in the array. MaxNum sets the limit on the number |

| of data nodes that the Heap can have. If you are only using |

| the heap for sorting, you can set MaxNum to Num. (However, |

| MaxNum should not be set less than Num). |

| |

| When the constructor is called, the Heap \*copies\* the Array. |

| Thus, neither the Array variable nor what it points to will |

| be modified. |

| |

| The second step is to call the actual sort: |

| |

| NewArray \*Sort(); |

| |

| This sort will return a pointer to another array, which is |

| the sorted array. Any modifications done to NewArray or its |

| contents will not affect the heap. |

| |

-------------------------------------------------------------------

\*/

#ifndef \_\_HeapTreeClassH\_\_

#define \_\_HeapTreeClassH\_\_

#include <assert.h> // For error-checking purposes

//-------------------------------------------------

// Main structure of HeapTree Class:

//-------------------------------------------------

template <class Elem>

class HeapTree

{

public:

HeapTree(int MaxSize=500);

HeapTree(const HeapTree<Elem> &OtherTree);

HeapTree(Elem \*Array, int ElemNum, int MaxSize);

Elem \*Sort(void); // Built-in HeapSort Algorithm

~HeapTree(void);

bool Add(const Elem &Item); // Add the Item to Heap

Elem Remove(void); // Remove and return Item from Heap

inline int GetSize(void); // Returns the number of nodes in the Heap

protected:

Elem \*Data; // Actual Data array

int CurrentNum; // Current number of elements

const int MAX\_SIZE; // The maximum number of elements

void ShiftUp(int Node); // Shift Node up into place

void ShiftDown(int Node); // Shift Node down into place

inline int ParentOf(int Node); // Returns Parent location

inline int LeftChildOf(int Node); // Returns Left Child location

};

//-------------------------------------------------

// Implementation of HeapTree Class:

//-------------------------------------------------

// HeapTree constructor function

template <class Elem>

HeapTree<Elem>::HeapTree(int MaxSize)

: MAX\_SIZE(MaxSize)

{

Data = new Elem[MAX\_SIZE];

CurrentNum = 0;

}

// HeapTree copy constructor function

template <class Elem>

HeapTree<Elem>::HeapTree(const HeapTree<Elem> &OtherTree)

: MAX\_SIZE(OtherTree.MAX\_SIZE)

{

Data = new Elem[MAX\_SIZE];

CurrentNum = OtherTree.CurrentNum;

// Copy the array

for (int i = 0; i < OtherTree.CurrentNum; ++i)

Data[i] = OtherTree.Data[i];

}

// HeapTree array constructor

template <class Elem>

HeapTree<Elem>::HeapTree(Elem \*Array, int ElemNum, int MaxSize)

: MAX\_SIZE(MaxSize)

{

Data = new Elem[MAX\_SIZE];

CurrentNum = ElemNum;

// This copies the array into the heap's internal array

for (int i = 0; i < ElemNum; ++i)

Data[i] = Array[i];

// This organizes the Array into a proper HeapTree

for (int i = ParentOf(CurrentNum - 1); i >= 0; --i)

ShiftDown(i);

}

// Built-in Heap Sort algorithm

template <class Elem>

Elem \*HeapTree<Elem>::Sort(void)

{

// This is the array that will be returned

Elem \*NewArray = new Elem[CurrentNum];

// The algorithm works back to front, with the sorted

// elements being stored in NewArray

for (int ElemNum = CurrentNum-1; ElemNum >=0; --ElemNum)

{

// Since the Remove() function alters CurrentNum by subtracting 1

// from it each time, we must use a seperate variable to

// index NewArray.

NewArray[ElemNum] = Remove();

}

return NewArray;

}

// HeapTree destructor function

template <class Elem>

HeapTree<Elem>::~HeapTree(void)

{

if (Data)

delete Data;

}

// Add() function

template <class Elem>

bool HeapTree<Elem>::Add(const Elem &Item)

{

if (CurrentNum >= MAX\_SIZE) // If we have reached our maximum capacity

return false;

Data[ CurrentNum ] = Item;

ShiftUp(CurrentNum++);

return true;

}

// Remove() function

template <class Elem>

Elem HeapTree<Elem>::Remove(void)

{

assert(CurrentNum > 0);

Elem Temp = Data[0];

Data[0] = Data[--CurrentNum]; // Replace with the last element

ShiftDown(0);

return Temp;

}

// GetSize() function

template <class Elem>

inline int HeapTree<Elem>::GetSize(void)

{

return CurrentNum;

}

// ShiftUp() function

template <class Elem>

void HeapTree<Elem>::ShiftUp(int Node)

{

int Current = Node,

Parent = ParentOf(Current);

Elem Item = Data[Current];

while (Current > 0) // While Current is not the RootNode

{

if (Data[Parent] < Item)

{

Data[Current] = Data[Parent];

Current = Parent;

Parent = ParentOf(Current);

}

else

break;

}

Data[Current] = Item;

}

// ShiftDown() function

template <class Elem>

void HeapTree<Elem>::ShiftDown(int Node)

{

int Current = Node,

Child = LeftChildOf(Current);

Elem Item = Data[Current]; // Used to compare values

while (Child < CurrentNum)

{

if (Child < (CurrentNum - 1))

if (Data[Child] < Data[Child+1]) // Set Child to largest Child node

++Child;

if (Item < Data[Child])

{ // Switch the Current node and the Child node

Data[Current] = Data[Child];

Current = Child;

Child = LeftChildOf(Current);

}

else

break;

}

Data[Current] = Item;

}

// ParentOf() function

template <class Elem>

inline int HeapTree<Elem>::ParentOf(int Node)

{

assert(Node > 0);

// This uses the fact that decimals are truncated during

// the division of integers. Thus, (12 - 1) / 2 == 5

return (Node - 1) / 2;

}

// LeftChildOf() function

template <class Elem>

inline int HeapTree<Elem>::LeftChildOf(int Node)

{

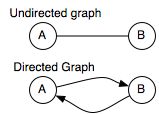
return (Node \* 2) + 1;

}

#endif /\*\_\_HeapTreeClassH\_\_\*/

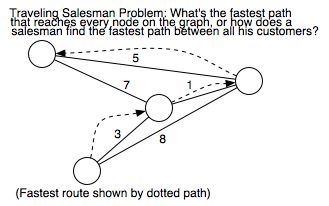
### Đồ thị

##### What is a graph?



A graph is a way of representing connections between places. Mathematically, a graph is a collection of nodes and edges. Nodes are locations that are connected together by the edges of the graph. For instance, if you had two small towns connected by a two-way road, you could represent this as a graph with two nodes, each node representing a town, and one edge, the road, connecting the two towns together. In addition to the undirected graph, in which the edge is a two-way connection, there are directed graphs, in which edges connect only one way. For instance, you could represent the previous example of two cities connected by a road as a directed graph consisting of two nodes and two edges, each edge connecting one of the nodes to the other. In the city example, it may also be convenient to record the distance between the two cities; this can be expressed by adding a 'weight' to an edge, which is a number that usually corresponds to the distance covered by an edge (the distance between two nodes).

##### Why are graphs useful?

Computer scientists have developed a great deal of theory about graphs and operations on them. One reason for this is because graphs can be used to represent many problems in computer science that are otherwise abstract. Finding a way to represent the solution to a problem as a graph can present new approaches to solving the problem or even lead directly to a solution derived from graph theory. This sort of technique is often used when discussing algorithmic efficiency and when trying to prove that a certain algorithm is NP-Complete; because many problems involving graphs, such as finding the shortest path to traverse all nodes (the Traveling Salesman Problem), are NP-Complete, if you can find a way to represent a problem as a graph and show that it is analogous to one of the other NP-Complete problems, then you can show the problem you are trying to solve is also NP-Complete, which gives you a hint that the solution will take a great deal of time.

Another reason for using graphs is that many problems computers are used to solve involve representing relationships between objects, places, or concepts. Because graphs can be either directed or undirected, they are a flexible method of showing connections. For instance, you can describe who knows whom in a room as a collection of nodes, each representing a person, and directed edges, each representing that one person knows another.

Because graphs are so often used and because they allow the representation of many problems in computer science, such as the Traveling Salesman Problem or something as simple as the relationships between people in a room, they are a convenient means of expressing problems with which many people are comfortable. This familiarity simplifies the process of creating mental models of problems, which ultimately leads to better problem solving.

##### What are some important graph theory terms?

Directed Graph: A directed graph is one in which edges connect nodes in only one direction.

Undirected Graph: An undirected graph is one in which edges connect nodes bidirectionally (in both directions). Node: A node, usually drawn as a circle, represents an item that can be related to other items or nodes. Nodes are sometimes referred to as vertices.

Edge: An edge represents a connection between nodes. Edges can be either directed or undirected, depending on the type of graph. Edges can also have weights, which may correspond to strength of relationship or distance between edges. (For instance, if a graph represents a map, then the weights of each edge will represent the distance between two nodes.)

Connected: A graph is connected if from any node you can reach any other node.

Disconnected: A graph is disconnected if certain groups of nodes form an island that has no connection to the rest of the graph.

## Các thuật toán

### Tính hiệu quả của giải thuật

##### Big-O và hiệu quả thuật toán

Chúng ta có nhiều giải pháp để giải quyết vấn đề, ví dụ phương pháp sắp xếp. Nhiều thuật toán khá nổi tiếng, nhưng vấn đề không phải là cách sắp xếp thế nào, mà là tính hiệu quả của phương pháp này. Chương này giúp bạn hiểu cách so sánh tính hiệu quả giữa các thuật toán và tại sao lại quan trọng để làm vậy.

Chúng ta dùng lý thuyết định giá trị Big- O để định nghĩa tính hiệu quả.

1. Tại sao Brute Force không đủ hiệu quả

Có một số vấn đề phải dùng brute force để giải quyết, như tìm tất cả khả năng kết hợp (như sắp xếp một cụm từ, tìm ra mọi khả năng đảo thứ tự cho tới khi tìm đúng thứ tự), vậy tại sao phải tìm cách tốt hơn? Câu trả lời đơn giản là, nếu bạn có máy tính đủ nhanh và số khả năng nhỏ, thì thời gian xử lí không thành vấn đề.

Nhưng để brute force cho một số lượng lớn các khả năng xảy ra, ví dụ 100 từ, tức có 100! Khả năng sắp xếp (tới 158 chữ số. Trong khi máy tính chỉ có thể tính toán 1 tỉ khả năng/giây, bạn cần 10^10\*149 giây để hoàn thành, ngang bằng thời gian của vũ trụ) thì yếu tố tốc độ xử lí không mang tính quyết định.

Rõ ràng là tính hiệu quả thuật toán cho việc sắp từ như trên phải tối ưu lại.

Trước hết, bạn làm quen với các thuật ngữ để đo tính hiệu quả thuật toán. Thông thường, tính hiệu quả được hiểu là thời gian bao lâu để xuất ra được kết quả.

ĐÁNH GIÁ THEO OUTPUT

Trong ví dụ trên, chúng ta biểu diễn giải thuật sơ cấp để sắp xếp từ thành một con số các khả năng. Thường ta xét độ dài input là n; tính hiệu quả trong ví dụ trên là n!.

Bạn để ý rằng, khả năng giải quyết vấn đề nhanh hơn là có thể- ví dụ, nếu các từ đã theo thứ tự một phần, thì không nhất thiết phải thử tới n! khả năng kết hợp.

Chúng ta hay nhắc tới khái niệm hiệu suất trung bình, trường hợp này là n!/2. Nhưng chia cho hai thì khá vô nghĩa khi n càng lớn thì n/2 cũng càng lớn, chúng ta thường bỏ qua các thuật ngữ hằng số (trừ khi nó là zero).

ĐÁNH GIÁ THEO INPUT

Bây giờ chúng ta mô tả mọi hiệu suất thuật toán theo nghĩa độ đài input (giả sử chúng ta đã biết cách tính hiệu suất), chúng ta có thể so sánh thuật toán dựa trên thứ bậc của chúng. “Bậc” ở đây hiểu là phương pháp toán học dùng so sánh hiệu suất- ví dụ, n^2 là bậc hai của n, và n! là “Thứ bậc của n giai thừa”. Hiển nhiên, thứ bậc của thuật toán n^2 thì ít hiệu quả hơn thuật toán có thứ bậc n. Nhưng không phải tất cả thứ bậc đều là đa thức – chúng ta đã thấy n!, và vài thuật toán sẽ là log n hay 2^n.

1. Ký hiệu Big-O

Theo nguyên tắc:

* Thứ bậc đại diện bởi kí hiệu O: ví dụ O(n^2). Gọi là big-O, là cách thông thường để diễn tả hiệu suất thuật toán.
* Big-O loại bỏ các yếu tố hằng số. Nếu bạn đang cần xác định thứ bậc thuật toán và nó là dạng đa thức, bạn phải biểu diễn biểu thức có thứ bậc cao nhất. Ví dụ: n^2 + n, thì thứ bậc là O(n^2).

Ngành khoa học máy tính dùng ký hiệu Big-O để nói về thuật toán, cũng như phân tích code. Phần kế tiếp, bạn sẽ học cách phân tích thuật toán và xác định chỉ số big-O.

##### Đánh giá thuật toán với Big-O

Kỹ năng phân tích code hay thuật toán và nắm được hiệu suất của nó là điều cốt lõi đối với ngành khoa học máy tính cũng như đảm bảo chương trình của bạn chạy nhanh mà không làm phiền người dùng.

PHƯƠNG PHÁP ĐỂ XÁC ĐỊNH SỐ BIG-O (THỨ BẬC THUẬT TOÁN):

Thứ bậc của O(1): diễn tả một thuật toán không làm gì cả, bất kể độ dài input.

* Tìm đoạn code có thứ bậc cao: xét từ bên ngoài đi vào trong
* Tìm hiệu suất của vòng lặp ngoài hay đệ qui hàm.
* Sau đó tìm hiệu suất của đoạn code bên trong.
* Tính tổng hiệu suất của mỗi tầng code với nhau.

Ví dụ đánh giá hiệu suất thuật toán selection sort:

for(int x=0; x<n; x++)

{

int min = x;

for(int y=x; y<n; y++)

{

if(array[y]<array[min])

min=y;

}

int temp = array[x];

array[x] = array[min];

array[min] = temp;

}

Chúng ta tính thứ bậc của vòng lặp ngoài là O(n).

Sau đó tính thứ bậc bên trong vòng lặp : làm tròn là O(n)

(Tính trung bình hiệu suất dựa trên giá trị x đi từ 0 tới n là: n/2, nhưng ta loại bỏ yếu tố hằng, nên là O(n)).

Cuối cùng tính tổng hiệu suất toàn cục, nhân các chỉ số big-O với nhau: O(n^2).

KHẢ NĂNG SUY LUẬN

Xác định chính xác thứ bậc đòi hỏi bạn phải có khả năng suy luận nhiều bưới của một thuật toán. Và bạn không phải lúc nào cũng soi từng đoạn code nhỏ, chỉ khi ta cần thảo luận về lý thuyết và xác định thứ bậc của nó

Một vài trường hợp cần ưu tiên hiệu suất hơn bất cứ thứ gì khác, vậy sang phần sau, bạn sẽ thấy danh sách đánh giá hiệu suất quan trọng và hữu ích, cùng với ví dụ giải thuật tương ứng.

##### Một số trường hợp cụ thể

1. O(1)

Thuật toán với big-O của O(1), chạy cùng tốc độ bất kể input. Ví dụ: giải thuật luôn trả về cùng một giá trị bất kể input có được xem xét hay không.

1. O(logn)

Thuật toán trên cây nhị phân thường có hiệu suất O(logn). Vì cây nhị phân tìm kiếm cân bằng hoàn hảo có logn tầng, và để tìm phần tử bất kì trên cây, phải yêu cầu duyệt qua từng node trên mỗi tầng

1. O(n)

Thuật toán này sẽ duyệt qua mọi phần tử n. Ví dụ về phương pháp tìm kiếm tuyến tính, tìm giá trị trên mảng bằng cách kiểm tra từ đầu tới cuối. Hay bạn phải truy xuất phần tử nào đó trên danh sách liên kết, vì DSLK không hỗ trợ truy xuất ngẫu nhiên như mảng.

1. O(nlogn)

Theo kinh nghiệm, thuật toán sắp xếp tốt luôn có hiệu suất xấp xỉ O(nlogn). Như merge sort, bẻ đôi mảng thành hai, sắp xếp từng phần bằng đệ qui, sau đó merging kết quả lại thành mảng đơn.

Vì nó cắt mảng làm đôi cho mỗi lần đệ qui, vòng lặp ngoài có hiệu suất là O(logn), tại mỗi cấp của mảng phân rã (thành hai, thành bốn…), nó sẽ hợp nhất lại đúng n phần tử, nên có hiệu suất O(n). Tổng kết lại là O(nlogn).

1. O(n^2)

Một hiệu suất tạm chấp nhận được, với khung thời gian đa dạng, như ví dụ về selection sort đề cập bên trên.

1. O(2^n)

Một hiệu suất phi đa thức quan trong khác đó là thời gian tăng theo cấp lũy thừa. Nhiều vấn đề quan trọng chỉ có thể giải quyết bởi thuật toán với hiệu suất này (hiệu suất tồi).

Một ví dụ là biểu diễn thừa số của một số lớn ở dạng nhị phân. Cách duy nhất thu nhặt phép dư từ phép chia từng thừa số cho hai liên tiếp. Mỗi khi có thêm một con số, sẽ tăng số vòng lặp gấp đôi.

1. Bảng so sánh

Bảng so sánh dưới đây giúp chúng ta dễ hình dung độ lớn của các mức thời gian thực hiện thuật giải nói trên:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Log n** | **n** | **n.log n** | **n2** | **2n** |
| 0 | 1 | 0 | 1 | 2 |
| 1 | 2 | 2 | 4 | 4 |
| 2 | 4 | 8 | 16 | 16 |
| 3 | 8 | 24 | 64 | 256 |
| 4 | 16 | 64 | 256 | 65536 |
| 5 | 32 | 160 | 1024 | 4,292,967,296 |

Rõ ràng là khi thời gian thực hiện thuật giải tăng với tốc độ hàm mũ thì độ lớn tăng rất nhanh. Những bài toán mà chưa tìm được thuật giải với thời gian dưới cấp hàm mũ, nghĩa là từ thời gian đa thức thức trở xuống, sẽ được xếp vào loại bài toán khó.

### Giải thuật đệ qui

Đệ quy (Recursion) là một trong những giải thuật khá quen thuộc trong lập trình, mở rộng ra là trong toán học (thường được gọi với tên khác là “quy nạp”). Có một số bài toán, buộc phải sử dụng đệ quy mới giải quyết được, chẳng hạn như duyệt cây.

1. Khái niệm

Đệ qui xảy ra khi một hàm tự gọi chính nó trực tiếp hay gián tiếp. Một vài hàm mượn chính nó để đệ qui giải thuật, như là tính giai thừa

Hàm đệ qui phải thỏa mãn 2 điều kiện:

1. Phải có điểm dừng (còn gọi là điểm neo) tại một trường hợp đặc biệt, gọi là trường hợp suy biến (degenerate case).

2. Phải đơn giản hóa được vấn đề

Sau khi đọc bài viết [MEMORY SEGMENT](http://www.stdio.vn/articles/read/25-memory-segment) , chúng ta hiểu rằng khi một hàm được gọi, nó sẽ được đưa vào **Stack**, hàm đệ quy cũng vậy, mỗi lần gọi chính nó thì nó lại được đưa vào Stack, nếu như không có điểm dừng, hoặc gọi mãi mà chưa tới điểm dừng, sẽ dễ xảy ra tình trạng tràn bộ nhớ Stack.

1. Hàm đệ quy

Ví dụ về tính gia thừa

fact(0) = 1

fact(n) = n \* fact(n-l)

Trong lập trình:

int fact(int number){

if (number == 0)

return (1);

else \*/

return (number \* fact(number-1));

}

Định nghĩa hàm đệ qui giai thừa thỏa 2 điều kiện:

* Đầu tiên, nó có điểm dừng khi number = 0.
* Thứ hai, nó đơn giản hóa bài toán với fact(number – 1)

Giai thừa hợp lệ phải có number >= 0. Nếu fact(-3), chương trình sẽ đệ qui vô hạn, báo stack overflow.

1. Thành phần của một hàm đệ quy

Hàm đệ quy gồm 2 phần:

* Phần cơ sở: Điều kiện thoát khỏi đệ quy
* Phần đệ quy: Thân hàm có chứa lời gọi đệ quy

1. Thiết kế giải thuật đệ quy

Thực hiện 3 bước sau:

* Tham số hóa bài toán
* Phân tích trường hợp chung: Đưa bài toán về bài toán nhỏ hơn cùng loại, dần dần tiến tới trường hợp suy biến
* Tìm trường hợp suy biến

Trong các hàm đệ qui, khi bạn cần phải khai báo biến trong quá trình tính toán, tốt nhất là khai báo biến static để tiết kiệm bộ nhớ stack, vì biến static chỉ được cấp phát vùng nhớ một lần, dù là đệ qui hàm.

1. Ưu và nhược điểm

Giải thuật đệ quy có ưu điểm là thuận lợi cho việc biểu diễn bài toán, đồng thời làm gọn chương trình. Tuy nhiên cũng có nhược điểm, đó là không tối ưu về mặt thời gian (so với sử dụng vòng lặp), gây tốn bộ nhớ.

1. Ví dụ khác

Rất nhiều thứ chúng ta phải làm đi làm lại có thể xử lí qua đệ qui, như tính tổng các phần tử mảng. Bạn có thể định nghĩa hàm cộng phần tử thứ “m” tới “n” của mảng như sau:

Coding:

int sum(int first, int last, int array[]){

if (first == last)

return (array[first]);

/\* else \*/

return (array[first] + sum(first + 1, last, array));

}

Chạy chương trình bằng tay:

Sum(1 8 3 2) =

1 + Sum(8 3 2) =

8 + Sum(3 2) =

3 + Sum(2) =

2

3 + 2 = 5

3+2=5

8 + 5 = 13

1 + 13 = 14

Kết quả = 14

1. Đệ qui với cấu trúc dữ liệu

Một vài cấu trúc mượn chính nó để thể hiện giải thuật đệ qui vì kết cấu dữ liệu được mô tả chứa một phiên bản nhỏ hơn cùng cấu trúc.

Vì đệ qui giải quyết vấn đề từ phiên bản nhỏ nhất, và nó làm việc tốt trên cấu trúc dữ liệu được tạo bởi nhiều cấu trúc nhỏ hơn – Danh sách liên kết (DSLK) cũng là cấu trúc dữ liệu.

Có thể xem DSLK được cấu thành bởi nhiều DSLK nhỏ hơn (một đoạn), hay bởi node đầu tiên trỏ tới DSLK đằng sau nó. DSLK tạo bởi nhiều node nhỏ, mỗi node lại trỏ tới node bắt đầu của DSLK còn lại.

Với đặc tính như trên, nó rất hữu ích để viết chương trình làm việc với DSLK, xử lí cả node hiện tại hay “phần còn lại DSLK”. Ví dụ: để tìm node cụ thể nào đó, bạn dùng giải thuật sau:

* Trường hợp suy biến: khi tới cuối DSLK, return NULL
* Nếu node hiện tại là cái cần tìm, return node
* Nếu không thì lặp tìm kiếm trên phần còn lại DSLK.

Coding như sau:

struct node{

int value;

node \*next;

};

node\* search (node\* list, int value\_to\_find){

if ( list == NULL ){

return NULL;

}

if ( list->value == value\_to\_find ){

return list;

}

else{

return search( list->next, value\_to\_find );

}

}

Hàm tìm kiếm được triển khai theo cách nói “ Nếu node hiện tại là cái cần tìm, trả về nó, nếu không thì giao kèo của hàm là phải tìm node trong danh sách – hay phần còn lại danh sách”.

Đệ qui có thể dùng theo hai cách – giải quyết vấn đề bằng các lệnh gọi đệ qui hay trả về hàm giải quyết vấn đề và dùng kết quả đó để tính toán thêm.

1. Một số loại đệ quy
2. Đệ quy tuyến tính (Linear Recursion)

Mỗi lần thực thi chỉ gọi đệ quy một lần

Một ví dụ rất quen thuộc là hàm tính giai thừa:

1. int Factorial(int n){
2. if (n == 0){
3. return 1;
4. }
5. else {
6. return n \* Factorial(n - 1); // Linear Recursion
7. }
8. }
9. Đệ quy nhị phân (Binary Recursion)

Mỗi lần thực thi có thể gọi đệ quy 2 lần

Ví dụ: Tính tổ hợp chập K của N bằng đệ quy:

1. int Combine(int n, int k)
2. {
3. if (k == 0 || k == n)
4. {
5. return 1;
6. }
7. else
8. {
9. return (Combine(n - 1, k) + Combine(n - 1, k - 1)); // Binary Recursion
10. }
11. }
12. Đệ quy lồng (Nested Recursion)

Tham số trong lời gọi đệ quy là một lời gọi đệ quy. Đệ quy lồng chiếm bộ nhớ rất nhanh.

Ví dụ: Hàm Ackerman

1. int Ackerman(int m, int n)
2. {
3. if (m == 0)
4. {
5. return (n + 1);
6. }
7. else
8. {
9. if (n == 0)
10. {
11. return Ackerman(m - 1, 1);
12. }
13. else
14. {
15. return Ackerman(m - 1, Ackerman(m, n - 1)); // Nested Recursion
16. }
17. }
18. }
19. Đệ quy hỗ tương (Mutual Recursion)

Các hàm gọi đệ quy lẫn nhau

Ví dụ: Xét tính chẵn lẻ của một số nguyên dương bằng đệ quy.

1. bool isEven(unsigned int n)
2. {
3. if (n == 0)
4. {
5. return true;
6. }
7. else
8. {
9. return isOdd(n - 1);
10. }
11. }
13. bool isOdd(unsigned int n)
14. {
15. if (n == 1)
16. {
17. return true;
18. }
19. else
20. {
21. return isEven(n - 1);
22. }
23. }

Để sử dụng đệ quy tương hỗ, cần phải khai báo Prototype. Tham khảo bài viết [FUNCTION PROTOTYPE](http://www.stdio.vn/articles/read/103-function-prototype).

1. Quay lui (Backtracking)

Trong lập trình, có một chiến lược giải bài toán một cách đầy đủ nhất, đảm bảo đủ mọi trường hợp bằng phương pháp Thử và Sai (Try and Error).

Nét đặc trưng của phương pháp này là ở chỗ các bước đi đến lời giải hoàn toàn bằng cách làm thử. Nếu có một lựa chọn được chấp nhận thì ghi nhớ các thông tin cần thiết các bước thử tiếp theo. Trái lại, nếu không có một lựa chọn nào thích hợp thì làm lại bước trước, xoá bớt các ghi nhớ và quay về chu trình thử với các lựa chọn còn lại. Hành động này được gọi là quay lui (Back tracking) và các giải thuật thể hiện phương pháp này gọi là các giải thuật quay lui.

Việc quay lui để thử tất cả các tổ hợp có thể có để đạt được lời giải cuối cùng.

Bài toán điển hình là liệt kê các cấu hình. Mỗi cấu hình được xây dựng bằng cách xây dựng từng phần tử, mỗi phần tử được chọn bằng cách thử tất cả các khả năng.

Để cho dễ hiểu ta xét bài toán cụ thể sau:

Bài toán N-Hậu (N-Queen): đặt N quân hậu lên trên một bàn cờ có kích thước NxN ô, sao cho không quân hậu nào ăn được quân hậu nào. Hay nói cách khác là không có hai quân hậu bất kì nào trên cùng một hàng, một cột hoặc một đường chéo.

Ta sử dụng đệ quy quay lui để giải bài toán như sau:

1. //N-Queen Problem in C++ Program
2. #include <iostream>
3. using namespace std;
5. #define QUEEN\_SIGN 'Q'
6. #define BLANK\_SIGN '\_'
8. #define N 8

11. //Set a Board with all Blank
12. void setBoardBlank(char Board[N][N])
13. {
14. for (int i = 0; i < N; i++)
15. {
16. for (int j = 0; j < N; j++)
17. {
18. Board[i][j] = BLANK\_SIGN;
19. }
20. }
21. }

24. //Print a Board to screen
25. void printBoard(char Board[N][N])
26. {
27. for (int i = 0; i < N; i++)
28. {
29. for (int j = 0; j < N; j++)
30. {
31. cout << Board[i][j];
32. }
33. cout << endl;
34. }
35. }

38. //Check Queen if at postion [RowIndex][ColumIndex] is Safe or not
39. bool isQueenSafeAtPosition(char Board[N][N], int RowIndex, int ColumnIndex)
40. {
41. //Check safe at Row and Column
42. for (int i = 0; i < N; i++)
43. {
44. if (Board[RowIndex][i] == QUEEN\_SIGN)
45. {
46. return false;
47. }
49. if (Board[i][ColumnIndex] == QUEEN\_SIGN)
50. {
51. return false;
52. }
53. }
55. //Check safe at first diagonal
56. for (int i = RowIndex, j = ColumnIndex; (i >= 0) && (j >= 0); i--, j--)
57. {
58. if (Board[i][j] == QUEEN\_SIGN)
59. {
60. return false;
61. }
62. }
64. for (int i = RowIndex, j = ColumnIndex; (i < N) && (j < N); i++, j++)
65. {
66. if (Board[i][j] == QUEEN\_SIGN)
67. {
68. return false;
69. }
70. }
72. //Check safe at second diagonal
73. for (int i = RowIndex, j = ColumnIndex; (i >= 0) && (j < N); i--, j++)
74. {
75. if (Board[i][j] == QUEEN\_SIGN)
76. {
77. return false;
78. }
79. }
81. for (int i = RowIndex, j = ColumnIndex; (i < N) && (j >= 0); i++, j--)
82. {
83. if (Board[i][j] == QUEEN\_SIGN)
84. {
85. return false;
86. }
87. }
89. return true;
90. }

93. //Solve N-Queen Problem, Backtracking here!
94. bool solveNQueenProblem(char Board[N][N], int RowIndex)
95. {
96. if (RowIndex >= N)
97. {
98. return true;
99. }
101. for (int j = 0; j < N; j++)
102. {
103. if (isQueenSafeAtPosition(Board, RowIndex, j) == true)
104. {
105. Board[RowIndex][j] = QUEEN\_SIGN;
106. if (solveNQueenProblem(Board, RowIndex + 1) == true)
107. {
108. return true;
109. }
110. else
111. {
112. Board[RowIndex][j] = BLANK\_SIGN;
113. }
114. }
115. }
117. return false;
118. }

121. //Entry Point
122. int main()
123. {
124. char Board[N][N];
126. setBoardBlank(Board);
128. if (solveNQueenProblem(Board, 0) == false)
129. {
130. cout << "Can't find any solution!" << endl;
131. }
132. else
133. {
134. printBoard(Board);
135. }
137. return 0;
138. }

### Các giải thuật sắp xếp

##### Bubble Sort và Shaker Sort

Trong các thuật toán sắp xếp, Bubble Sort và Shaker Sort là hai thuật toán có nhiều nét tương đồng. Trong bài viết này tôi sẽ giới thiệu hai thuật toán này, đồng thời so sánh hai thuật toán để các bạn có cái nhìn toàn diện hơn. Trong bài viết, tôi chọn mảng số nguyên để minh hoạ trực quan hơn và sẽ sắp xếp theo chiều tăng dần.

1. Bubble Sort
2. Ý tưởng

Giả sử dãy ban đầu cần sắp xếp có N phần tử a1, a2, a3, …, aN. Ta sẽ xét lần lượt 2 phần tử liền kề nhau, bắt đầu từ cuối dãy số. Nếu 2 phần tử này là nghịch thế của nhau (a[i] > a[i – 1]) thì ta tiến hành hoán vị đổi chỗ hai phần tử này. Cứ như thế, ta sẽ đem được phần tử nhỏ nhất lên đầu dãy và loại phần tử này ra khỏi dãy hiện hành. Tiếp tục thực hiện lại các bước trên đối với dãy mới để đưa dần các phần tử nhỏ nhất về đầu dãy. Chúng ta sẽ thực hiện lại các thao tác trên cho đến khi dãy hiện hành chỉ còn 1 phần tử duy nhất.

[http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_4/117/BubbleSort.gif](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_4/117/BubbleSort.gif)

1. Hiện thực
2. void BubbleSort(int a[], int n)
3. {
4. for (int i = 0; i < n - 1; i++)
5. for (int j = n - 1; j > i; j--)
6. if (a[j] < a[j - 1])
7. swap(a[j], a[j - 1]);
8. }

Đánh giá

* Trong trường hợp tốt nhất đọ phức tạp là O(n).
* Độ phức tạp trong trường hợp trung bình là O(n2).
* Trong trường hợp xấu nhất đọ phức tạp là O(n2).
* Không nhận biết được mảng đã được sắp xếp.

1. Shaker Sort
2. Ý tưởng

Shaker Sort là một cải tiến của Bubble Sort. Sau khi đưa phần tử nhỏ nhất về đầu dãy, thuật toán sẽ giúp chúng ta đưa phần tử lớn nhất về cuối dãy. Do đưa các phần tử về đúng vị trí ở cả hai đầu nên Shaker Sort sẽ giúp cải thiện thời gian sắp xếp dãy số.

[http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_4/117/ShakerSort.gif](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_4/117/ShakerSort.gif)

1. Hiện thực
2. void ShakerSort(int a[], int n)
3. {
4. int Left = 0;
5. int Right = n - 1;
6. int k = 0;
7. while (Left < Right)
8. {
9. for (int i = Left; i < Right; i++)
10. {
11. if (a[i] > a[i + 1])
12. {
13. swap(a[i], a[i + 1]);
14. k = i;
15. }
16. }
17. Right = k;
18. for (i = Right; i > Left; i--)
19. {
20. if (a[i] < a[i - 1])
21. {
22. swap(a[i], a[i - 1]);
23. k = i;
24. }
25. }
26. Left = k;
27. }
28. }
29. Đánh giá

Như trên tôi đã nói Shaker Sort là một dạng nâng cao của Bubble Sort nên nó có thể nhận biết được mảng đã được sắp xếp.

* Độ phức tạp cho trường hợp tốt nhất là O(n).
* Độ phức tạp cho trường hợp xấu nhất O(n2).
* Độ phức tạp trong trường hợp trung bình là O(n2).
* Thuật toán nhận diện được mảng đã sắp xếp.

1. Kết luận

Trong trường hợp mảng có các phần tử là [2, 3, 4, 5, 1] thì đối với Shaker Sort chỉ cần 1 lần duyệt là đã đưa các phần tử của mảng về đúng vị trí, còn với Bubble Sort cần tới 4 lần duyệt để đưa các phần tử về đúng vị trí. Tuy nhiên, trong trường hợp mảng có ngẫu nhiên phần tử với thứ tự đảo lộn thì Bubble Sort và Shaker Sort cho thời gian sắp xếp gần tương đương nhau.

Vì vậy, ta có thể nói rằng Shaker Sort ưu thế hơn Bubble Sort trong trường hợp các phần tử trong mảng đã gần có thứ tự như trong ví dụ trên là mảng [2, 3, 4, 5, 1].

##### Selection Sort

Giới thiệu và hiện thực thuật toán sắp xếp Selection Sort. Trong bài này, tôi chỉ hiện thực việc xếp tăng dần, bạn có thể làm điều tương tự cho việc xếp giảm dần.

1. Ý tưởng

Chọn phần tử nhỏ nhất đưa về vị trí đầu tiên của dãy hiện tại và không cần quan tâm đến nó nữa, khi đó dãy chỉ còn lại n-1 phần tử của dãy ban đầu, lúc đó dãy ta xét sẽ bắt đầu từ phần tử thứ 2 của mảng, chúng ta lập lại cho đến khi dãy hiện tại chỉ còn 1 phần tử.



1. Các bước thực hiện

Bước 1: i = 0.  
Bước 2: Tìm phần tử a[iMin] trong dãy hiện hành từ a[i] đến a[n-1].  
Bước 3: Đổi chỗ a[i] và a[iMin].  
Bước 4: Nếu i < n-1 thì lặp lại bước 2 với i++ - Ngược lại thì dừng.

1. Hiện thực
2. void Selection(int a[], int n)
3. {
4. for (int i = 0; i < n - 1; i++)
5. {
6. int iMin = i;
7. for (int j = i + 1; j < n; j++)
8. {
9. if (a[iMin] > a[j])
10. iMin = j;
11. }
12. if (i != iMin)
13. swap(a[i], a[iMin]);
14. }
15. }
16. Ưu và nhược điểm

Số lần so sánh trong trường hợp tốt nhất là **n(n-1)/2**  
Số lần so sánh trong trường hợp xấu nhất là **3n(n-1)/2**

1. Ưu điểm

Thuật toán đơn giản, dễ hiện thực.

Có số lần hoán đổi các vị trí ít.

1. Nhược điểm

Chỉ được áp dụng trong các trường hợp có số lượng phần tử cần so sánh ít.

Không nhận biết được mảng đã được sắp xếp.

##### Insertion Sort

Ở bài viết trước, tôi đã giới thiệu với các bạn một thuật toán sắp xếp đơn giản, dễ hiện thực và sử dụng là [Selection Sort](http://www.stdio.vn/articles/read/110-selection-sort). Trong bài viết này, tôi sẽ tiếp tục giới thiệu và hiện thực thuật toán sắp xếp khác là Insertion Sort - sắp xếp chèn. Trong phạm vi bài viết, tôi chỉ hiện thực việc sắp xếp tăng dần, bạn có thể làm tương tự cho việc sắp xếp giảm dần và tôi sử dụng mảng các số nguyên để minh hoạ cho giải thuật.

1. Ý tưởng

Xét danh sách con gồm **k** phần tử đầu **a1 … ak**. Với **k = 1**, danh sách gồm một phần tử đã được sắp xếp thành dãy tăng dần. Giả sử trong danh sách **k-1** phần tử đầu **a1 … ak-1** đã được sắp xếp. Để sắp xếp phần tử **ak = x** ta tìm vị trí thích hợp của nó trong dãy **a1 … ak-1**. Vị trí thích hợp cần tìm là vị trí đứng trước phần tử lớn hơn nó và sau phần tử nhỏ hơn hoặc bằng nó.

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2014/quarter_4/112/ss_1.gif)

1. Hiện thực
2. void InsertionSort(int a[], int n)
3. {
4. for (int i = 1; i < n; i++)
5. {
6. int x = a[i];
7. int j = i;
8. while (j > 0 && a[j - 1] > x)
9. {
10. a[j] = a[j - 1];
11. j--;
12. }
13. a[j] = x;
14. }
15. }
16. Đánh giá

* Trong trường hợp tốt nhất thuật toán sữ dụng n-1 phép so sánh và 0 lần hoán vị.
* Trung bình thuật toán sử dụng n2/4 phép so sánh và n2/4 lần hoán vị.
* Trong trường hợp xấu nhất thuật toán sử dụng n2/2 phép so sánh và n2/2 lần hoán vị.
* Thuật toán thích hợp đối với mảng đã được sắp xếp một phần hoặc mảng có kích thước nhỏ.

##### Merge Sort

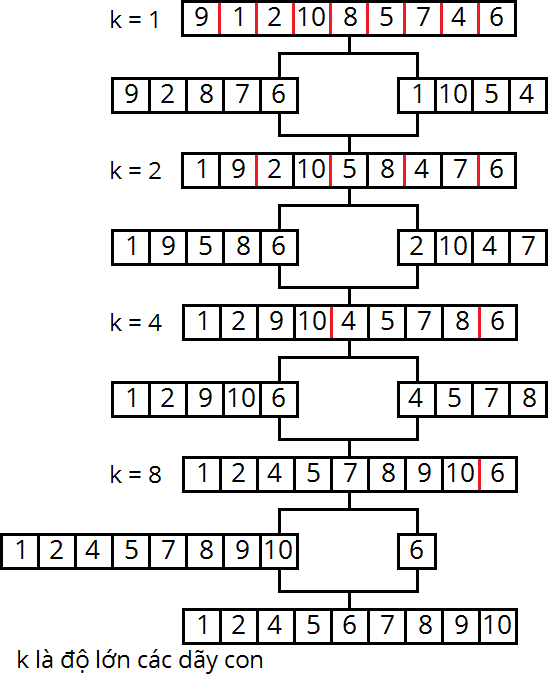
Trong các thuật toán sắp xếp, Merge sort là thuật toán không quá khó nhưng đạt hiệu quả cao trong nhiều trường hợp. Nhận thấy được hiệu quả mà nó mang lại, tác giả xin được sử dụng vốn kiến thức mà mình tìm hiểu được để hiện thực lại thuật toán này.

1. Ý tưởng

Thuật toán Merge Sort là loại thuật toán sắp xếp theo phương pháp trộn có độ phức tạp trung bình (O(n.logn)).

Dãy cần sắp xếp sẽ có những đoạn dãy con đã có thứ tự tăng dần hoặc giảm dần. Thuật toán làm giảm dần các dãy con, chỉ còn một dãy con duy nhất có chiều tăng giảm đúng với yêu cầu người viết bằng cách chia dãy ban đầu ra 2 dãy phụ theo nguyên tắc luân phiên, cứ một phần tử vào dãy thứ nhất thì phần tử tiếp sau vào dãy thứ hai và ngược lại.

Trộn từng cặp dãy con trong hai dãy phụ vào dãy ban đầu ta sẽ được dãy mới có các phần tử tương tự dãy ban đầu nhưng có trật tự thay đổi, số đoạn dãy con giảm dần, ít nhất là giảm đi một nửa theo hướng sắp xếp của người viết. Tuần tự tăng dần độ lớn của dãy con và thực hiện tương tự các bước trên cho đến khi độ lớn của dãy con vượt quá số phần tử của dãy ban đầu.

Nếu bạn chưa hiểu tư tưởng của Merge sort. Đừng lo lắng, chúng ta sẽ đi đến ví dụ hình ảnh sau.  
  
[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_2/234/Untitled.png)

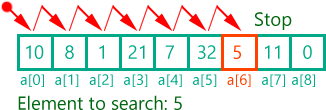
1. Hiện thực
2. #include <stdio.h>
4. // trả về số nhỏ hơn
5. int min(int a, int b)
6. {
7. if (a < b)
8. return a;
9. return b;
10. }
12. // trộn 2 dãy phụ tạo dãy mới
13. void merge(int arr[], int temp1[], int n1, int temp2[], int n2, int k)
14. {
15. int i1, i2, i;
16. int k1, k2;
17. int j1, j2;
18. i = i1 = i2 = 0;
19. j1 = j2 = 0;
21. while (n2 > 0 && n2 > 0)
22. {
23. // xác định độ dài từng dãy con 2 dãy phụ
24. k1 = min(k, n1);
25. k2 = min(k, n2);
27. // xét và trộn dãy con vào dãy
28. if (temp1[i1 + j1] < temp2[i2 + j2])
29. {
30. arr[i++] = temp1[i1 + j1];
31. j1++;
33. // trộn dãy con còn lại vào dãy
34. if (j1 == k1)
35. {
36. for (; j2 < k2; j2++)
37. arr[i++] = temp2[i2 + j2];
38. i1 += k1; i2 += k2;
39. j1 = j2 = 0;
40. n1 -= k1; n2 -= k2;
41. }
42. }
43. else
44. {
45. arr[i++] = temp2[i2 + j2];
46. j2++;
48. // trộn dãy con còn lại vào dãy
49. if (j2 == k2)
50. {
51. for (; j1 < k1; j1++)
52. arr[i++] = temp1[i1 + j1];
53. i1 += k1; i2 += k2;
54. j1 = j2 = 0;
55. n1 -= k1; n2 -= k2;
56. }
57. }
58. }
59. }
61. void mergeSort(int arr[], int n)
62. {
63. int n1, n2;
64. int i;
65. int k;
66. int ik;
67. int \*temp1 = new int[n];
68. int \*temp2 = new int[n];
69. k = 1;
71. do
72. {
73. i = n1 = n2 = 0;
75. // chia mảng ra 2 mảng phụ
76. while (i < n)
77. {
78. ik = 0;
80. while (ik < k && i < n)
81. {
82. temp1[n1++] = arr[i++];
83. ik++;
84. }
86. ik = 0;
88. while (ik < k && i < n)
89. {
90. temp2[n2++] = arr[i++];
91. ik++;
92. }
93. }
95. merge(arr, temp1, n1, temp2, n2, k);
97. // tăng độ lớn tối đa dãy con
98. k \*= 2;
99. } while (k < n);
101. delete[] temp1;
102. delete[] temp2;
103. }
105. void main()
106. {
107. int i, n;
108. int \*Array;
110. printf("How many elements do you want to sort? ");
111. scanf("%d", &n);
113. Array = new int[n];
115. for (i = 0; i < n; i++)
116. {
117. printf("Array[%d] = ", i);
118. scanf("%d", &Array[i]);
119. }
121. mergeSort(Array, n);
123. for (i = 0; i < n; i++)
124. {
125. printf("%d ", Array[i]);
126. }
128. delete[] Array;
129. }

##### Linear Search - Tìm Kiếm Tuyến Tính

Tìm kiếm là một nhu cầu thiết yếu trong cuộc sống, và đương nhiên, cũng không phải là ngoại lệ trong lập trình. Chúng ta có rất nhiều thuật toán để thực hiện thao tác tìm kiếm. Để mở đầu cho chuỗi bài viết về các thuật toán tìm kiếm, tôi sẽ bắt đầu với thuật toán tìm kiếm đơn giản nhất: Sequential search – Tìm kiếm tuần tự hay Linear search – Tìm kiếm tuyến tính.

1. Ý tưởng

Bạn hãy tưởng tượng rằng, bạn có một chiếc chìa khóa và một tập hợp các ổ khóa. Nhiệm vụ của bạn là phải kiếm xem có hay không ổ khóa vừa chiếc chìa khóa trên tay. Cách đơn giản nhất đó là, chúng ta sẽ thử chiếc chìa khóa với từng ổ khóa cho tới khi nhận được kết quả.

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_1/123/ss_1.png)

1. Hiện thực
2. bool LinearSearch(int arr[], int numberOfElements, int key, int& position)
3. {
4. int index;
6. for (index = 0; index < numberOfElements; ++index)
7. {
8. if(arr[index] == key)
9. {
10. break;
11. }
12. }
14. if(index < numberOfElements)
15. {
16. position = index;
17. return true;
18. }
20. return false;
21. }

Ở đây, tôi đưa ra một mảng arr gồm 9 phần tử, và cần tìm kiếm phần tử có giá trị bằng key = 5 trong mảng này.

1. Các bước thực hiện

* Bước 1: index = 0.
* Bước 2: Thực hiện so sánh arr[index] và key.  Nếu arr[index] == key thì dừng. Đưa ra vị trí của key trong mảng đồng thời trả về true (tìm thấy).
* Bước 3: Nếu index < number\_of\_element, thì lặp lại bước 2 với ++index. Ngược lại thì dừng và trả về false (không tìm thấy).

1. Đánh giá

* Trong trường hợp tốt nhất, phần tử cần tìm nằm ở vị trí đầu tiên, thuật toán sử dụng 1 lần so sánh.
* Trong trường hợp xấu nhất, phần từ cần tìm nằm ở vị trí cuối, hoặc không thuộc trong mảng, thuật toán sử dụng n-1 lần so sánh.
* Linear Search là một giải thuật đơn giản khi hiện thực và khá hiệu quả với danh sách đủ nhỏ hoặc một danh sách chưa được sắp xếp đơn giản.

##### Thuật toán Depth First Search

Bài viết giới thiệu, nêu khái quát và trình bày về thuật toán Depth First Search (DFS – Tìm kiếm theo chiều sâu), cùng với thuật toán Breadth First Search là hai thuật toán cơ bản để chuẩn bị ra các thuật toán phức tạp hơn trong bộ môn Trí tuệ nhân tạo.

Thuật toán Depth First Search (DFS – Tìm kiếm theo chiều sâu) là một dạng thuật toán duyệt hoặc tìm kiếm trên cây hoặc đồ thị. Trong lý thuyết khoa học máy tính, thuật toán DFS nằm trong chiến lược tìm kiếm mù (tìm kiếm không có định hướng, không chú ý đến thông tin, giá trị được duyệt) được ứng dụng để duyệt hoặc tìm kiếm trên đồ thị.

1. Ý tưởng thuật toán

Từ đỉnh (nút) gốc ban đầu, thuật toán duyệt đi xa nhất theo từng nhánh, khi nhánh đã duyệt hết, lùi về từng đỉnh để tìm và duyệt những nhánh tiếp theo. Quá trình duyệt chỉ dừng lại khi tìm thấy đỉnh cần tìm hoặc tất cả đỉnh đều đã được duyệt qua.

1. Thuật giải

Trước khi bắt đầu, tôi sẽ nêu ra một số quy ước để dễ dàng trong trình bày:

* Open: là tập hợp các đỉnh chờ được xét ở bước tiếp theo theo ngăn xếp (ngăn xếp: dãy các phần tử mà khi thêm phần tử vào sẽ thêm vào đầu dãy, còn khi lấy phần tử ra sẽ lấy ở phần tử đứng đầu dãy).
* Close: là tập hợp các đỉnh đã xét, đã duyệt qua.
* s: là đỉnh xuất phát, đỉnh gốc ban đầu trong quá trình tìm kiếm.
* g: đỉnh đích cần tìm.
* p: đỉnh đang xét, đang duyệt.

Trình bày thuật giải:

* Bước 1: Tập Open chứa đỉnh gốc s chờ được xét.
* Bước 2: Kiểm tra tập Open có rỗng không.
  + Nếu tập Open không rỗng, lấy một đỉnh ra khỏi tập Open làm đỉnh đang xét p. Nếu p là đỉnh g cần tìm, kết thúc tìm kiếm.
  + Nếu tập Open rỗng, tiến đến bước 4.
* Bước 3: Đưa đỉnh p vào tập Close, sau đó xác định các đỉnh kề với đỉnh p vừa xét. Nếu các đỉnh kề không thuộc tập Close, đưa chúng vào đầu tập Open. Quay lại bước 2.
* Bước 4: Kết luận không tìm ra đỉnh đích cần tìm.

1. Code

Tôi sẽ hiện thực thuật toán này theo ma trận kề bằng ngôn ngữ C++.

1. #include <stdio.h>
2. #include <stack>
3. using namespace std;
5. // dinh nghia lop do thi
6. class Graph
7. {
8. private:
9. int n;
10. int \*\*edge;
11. public:
12. Graph(int size = 2);
13. ~Graph();
14. bool isConnected(int, int);
15. void addEdge(int x, int y);
16. void depthFirstSearch(int, int);
17. };
19. Graph::Graph(int size)
20. {
21. int i, j;
23. // xac dinh so dinh cua do thi
24. if (size < 2)
25. n = 2;
26. else
27. n = size;
29. // tao ra cac dinh trong do thi
30. edge = new int\*[n];
31. for (i = 0; i < n; i++)
32. edge[i] = new int[n];
34. // mac dinh giua cac dinh khong co ket noi voi nhau (= 0)
35. for (i = 0; i < n; i++)
36. for (j = 0; j < n; j++)
37. edge[i][j] = 0;
38. }
40. Graph::~Graph()
41. {
42. for (int i = 0; i < n; ++i)
43. delete[] edge[i];
44. delete[] edge;
45. }
47. // kiem tra giua hai dinh co ke nhau hay khong
48. bool Graph::isConnected(int x, int y)
49. {
50. if (edge[x - 1][y - 1] == 1)
51. return true;
53. return false;
54. }
56. // thêm cạnh nối đỉnh x và đỉnh y
57. void Graph::addEdge(int x, int y)
58. {
59. if (x < 1 || x > n || y < 1 || y > n)
60. return;
62. edge[x - 1][y - 1] = edge[y - 1][x - 1] = 1;
63. }
65. void Graph::depthFirstSearch(int s, int g)
66. {
67. if (s > n || s < 0 || g > n || g < 0)
68. {
69. printf("Could not traverse this graph with your request\n");
70. return;
71. }
73. stack <int> open;
74. bool \*close = new bool[n];
75. int i;
76. int p;
78. // mac dinh cac dinh chua duoc duyet
79. for (i = 0; i < n; i++)
80. close[i] = false;
82. // dua dinh goc s vao stack open, chuan bi duyet
83. open.push(s);
85. printf("With Depth first Search , we have vertex(s):\n");
87. while (!open.empty())
88. {
89. // lay mot dinh ra khoi open tro thanh dinh dang xet p
90. do
91. {
92. if (open.empty())
93. return;
95. p = open.top();
96. open.pop();
97. } while (close[p - 1] == true);
99. // in ra dinh dang xet
100. printf("%d ", p);
102. // p da duyet qua
103. close[p - 1] = true;
105. // ket thuc duyet khi tim ra ket qua can tim
106. if (p == g)
107. return;
109. // tim dinh ke voi dinh dang xet, dinh nao chua duoc duyet dua vao open
110. for (i = 1; i <= n; i++)
111. {
112. if (isConnected(p, i) && !close[i - 1])
113. {
114. open.push(i);
115. }
116. }
117. }
118. printf("\n");
120. delete[] close;
121. }
122. Ví dụ

Tôi có một đồ thị như hình vẽ.



Tôi khởi tạo một đồ thị có 8 đỉnh và có cạnh nối các đỉnh kề như hình vẽ.

1. void main()
2. {
3. // khoi tao do thi
4. Graph g(8);
6. // tao canh noi giua cac dinh ke
7. g.addEdge(1, 2);
8. g.addEdge(1, 3);
9. g.addEdge(1, 4);
10. g.addEdge(1, 5);
11. g.addEdge(2, 6);
12. g.addEdge(3, 4);
13. g.addEdge(3, 8);
14. g.addEdge(4, 8);
15. g.addEdge(5, 8);
16. g.addEdge(6, 7);
17. g.addEdge(6, 8);
18. }

Tôi chọn đỉnh gốc để duyệt là đỉnh 1 và cần tìm đỉnh 4, tôi thực hiện duyệt ngay sau khi tạo cạnh nối giữa các đỉnh kề.

1. void main()
2. {
3. // khoi tao do thi
4. Graph g(8);
6. // tao canh noi giua cac dinh ke
7. g.addEdge(1, 2);
8. g.addEdge(1, 3);
9. g.addEdge(1, 4);
10. g.addEdge(1, 5);
11. g.addEdge(2, 6);
12. g.addEdge(3, 4);
13. g.addEdge(3, 8);
14. g.addEdge(4, 8);
15. g.addEdge(5, 8);
16. g.addEdge(6, 7);
17. g.addEdge(6, 8);
19. // duyet bang Depth First Search
20. g.depthFirstSearch(1, 4);
21. }

Bảng bên dưới thể hiện quá trình duyệt từ đỉnh 1 đến đỉnh 4.

| **Đỉnh đang xét** | **Các đỉnh kề** | **Open** | **Close** |
| --- | --- | --- | --- |
|  |  | {1} | {} |
| 1 | {2; 3; 4; 5} | {5; 4; 3; 2} | {1} |
| 5 | {1; 8} | {8; 4; 3; 2} | {1; 5} |
| 8 | {3; 4; 5; 6} | {6; 4; 3; 4; 3; 2} | {1; 5; 8} |
| 6 | {2; 7; 8} | {7; 2; 4; 3; 4; 3; 2} | {1; 5; 8; 6} |
| 7 | {6} | {2; 4; 3; 4; 3; 2} | {1; 5; 8; 6; 7} |
| 2 | {1; 6} | {4; 3; 4; 3; 2} | {1; 5 ;8 ;6; 7; 2} |
| 4 | {1; 3; 8} | {3; 4; 3; 3; 2} | {1; 5; 8; 6; 7; 2; 4} |

1. Ưu điểm

* Xét duyệt tất cả các đỉnhđể trả về kết quả.
* Nếu số đỉnh là hữu hạn, thuật toán chắc chắn tìm ra kết quả.

1. Khuyết điểm

* Mang tính chất vét cạn, không nên áp dụng nếu duyệt số đỉnh quá lớn.
* Mang tính chất mù quáng, duyệt tất cả đỉnh, không chú ý đến thông tin trong các đỉnh để duyệt hiệu quả, dẫn đến duyệt qua các đỉnh không cần thiết.

##### Thuật toán Breadth First Search

Bài viết giới thiệu, nêu khái quát và trình bày về thuật toán Breadth First Search (BFS – Tìm kiếm theo chiều rộng), cùng với thuật toán Depth First Search là hai thuật toán cơ bản để chuẩn bị ra các thuật toán phức tạp hơn trong bộ môn Trí tuệ nhân tạo.

Thuật toán BFS là thuật toán xét (duyệt) hoặc tìm kiếm trên cây và đồ thị, có chiến lược tìm kiếm mù (tìm kiếm không có định hướng, không chú ý đến thông tin, giá trị được duyệt).

1. Ý tưởng thuật toán

Từ một đỉnh (nút) gốc ban đầu là đỉnh đang xét, xác định và lần lượt duyệt các đỉnh kề xung quanh đỉnh gốc vừa xét. Tiếp tục quá trình duyệt qua các đỉnh kề đỉnh vừa xét cho đến khi đạt được kết quả cần tìm hoặc duyệt qua tất cả các đỉnh.

1. Thuật giải

Trước khi bắt đầu, tôi sẽ nêu ra một số quy ước để dễ dàng trong trình bày:

* Open: là tập hợp các đỉnh chờ được xét ở bước tiếp theo theo hàng đợi (hàng đợi: dãy các phần tử mà khi thêm phần tử vào sẽ thêm vào cuối dãy, còn khi lấy phần tử ra sẽ lấy ở phần tử đứng đầu dãy).
* Close: là tập hợp các đỉnh đã xét, đã duyệt qua.
* s: là đỉnh xuất phát, đỉnh gốc ban đầu trong quá trình tìm kiếm.
* g: đỉnh đích cần tìm.
* p: đỉnh đang xét, đang duyệt.

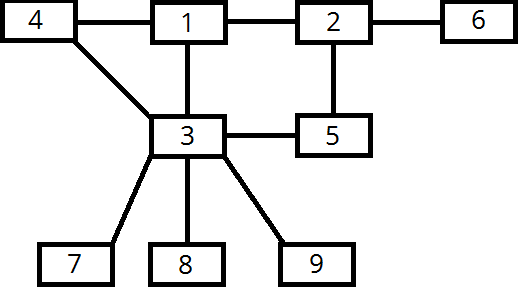
Trình bày thuật giải:

* Bước 1: Tập Open chứa đỉnh gốc s chờ được xét.
* Bước 2: Kiểm tra tập Open có rỗng không.
  + Nếu tập Open không rỗng, lấy một đỉnh ra khỏi tập Open làm đỉnh đang xét p. Nếu p là đỉnh g cần tìm, kết thúc tìm kiếm.
  + Nếu tập Open rỗng, tiến đến bước 4.
* Bước 3: Đưa đỉnh p vào tập Close, sau đó xác định các đỉnh kề với đỉnh p vừa xét. Nếu các đỉnh kề không thuộc tập Close, đưa chúng vào cuối tập Open. Quay lại bước 2.
* Bước 4: Kết luận không tìm ra đỉnh đích cần tìm.

1. Code

Tôi sẽ hiện thực thuật toán này theo ma trận kề bằng ngôn ngữ C++.

1. #include <stdio.h>
2. #include <queue>
3. #include <conio.h>
4. using namespace std;
6. // dinh nghia lop do thi
7. class Graph
8. {
9. private:
10. int n;
11. int \*\*edge;
12. public:
13. Graph(int size = 2);
14. ~Graph();
15. bool isConnected(int, int);
16. void addEdge(int, int);
17. void breadthFirstSearch(int, int);
18. };
20. Graph::Graph(int size)
21. {
22. int i, j;
24. // xac dinh so dinh cua do thi
25. if (size < 2)
26. n = 2;
27. else
28. n = size;
30. // tao ra cac dinh trong do thi
31. edge = new int\*[n];
32. for (i = 0; i < n; i++)
33. edge[i] = new int[n];
35. // mac dinh giua cac dinh khong co ket noi voi nhau (= 0)
36. for (i = 0; i < n; i++)
37. for (j = 0; j < n; j++)
38. edge[i][j] = 0;
39. }
41. Graph::~Graph()
42. {
43. for (int i = 0; i < n; ++i)
44. delete[] edge[i];
45. delete[] edge;
46. }
48. // kiem tra giua hai dinh co ke nhau hay khong
49. bool Graph::isConnected(int x, int y)
50. {
51. if (edge[x - 1][y - 1] == 1)
52. return true;
54. return false;
55. }
57. // tao canh noi giua hai dinh, lam cho hai dinh ke nhau
58. void Graph::addEdge(int x, int y)
59. {
60. if (x < 1 || x > n || y < 1 || y > n)
61. return;
63. edge[x - 1][y - 1] = edge[y - 1][x - 1] = 1;
64. }
66. void Graph::breadthFirstSearch(int s, int g)
67. {
68. if (s > n || s < 0 || g > n || g < 0)
69. {
70. printf("Could not traverse this graph with your request\n");
71. return;
72. }
73. queue <int> open;
74. bool \*close = new bool[n];
75. int i;
76. int p;
78. // mac dinh cac dinh chua duoc duyet
79. for (i = 0; i < n; i++)
80. close[i] = false;
82. // dua dinh goc s vao queue open, chuan bi duyet
83. open.push(s);
85. printf("With Breadth first Search , we have vertex(s):\n");
87. while (!open.empty())
88. {
89. // lay mot dinh ra khoi open tro thanh dinh dang xet p
90. do
91. {
92. if (open.empty())
93. return;
95. p = open.front();
96. open.pop();
97. } while (close[p - 1] == true);
99. // in ra dinh dang xet
100. printf("%d ", p);
102. // p da duyet qua
103. close[p - 1] = true;
105. // ket thuc duyet khi tim ra ket qua can tim
106. if (p == g)
107. return;
109. // tim dinh ke voi dinh dang xet, dinh nao chua duoc duyet dua vao open
110. for (i = 1; i <= n; i++)
111. {
112. if (isConnected(p, i) && !close[i - 1])
113. {
114. open.push(i);
115. }
116. }
117. }
118. printf("\n");
120. delete[] close;
121. }
122. Ví dụ

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_3/285/BFS.png)

Tôi khởi tạo đồ thị có 9 đỉnh và tạo cạnh nối các đỉnh kề nhau trong hàm main().

1. void main()
2. {
3. // khoi tao do thi
4. Graph g(9);
6. // tao canh noi giua cac dinh ke
7. g.addEdge(1, 2);
8. g.addEdge(1, 3);
9. g.addEdge(1, 4);
10. g.addEdge(2, 5);
11. g.addEdge(2, 6);
12. g.addEdge(3, 4);
13. g.addEdge(3, 5);
14. g.addEdge(3, 7);
15. g.addEdge(3, 8);
16. g.addEdge(3, 9);
17. }

Tôi chọn đỉnh gốc để duyệt là đỉnh 1 và cần tìm đỉnh 8, tôi thực hiện duyệt ngay sau khi tạo cạnh nối giữa các đỉnh kề.

1. void main()
2. {
3. // khoi tao do thi
4. Graph g(9);
6. // tao canh noi giua cac dinh ke
7. g.addEdge(1, 2);
8. g.addEdge(1, 3);
9. g.addEdge(1, 4);
10. g.addEdge(2, 5);
11. g.addEdge(2, 6);
12. g.addEdge(3, 4);
13. g.addEdge(3, 5);
14. g.addEdge(3, 7);
15. g.addEdge(3, 8);
16. g.addEdge(3, 9);
18. // duyet bang Breadth First Search
19. g.breadthFirstSearch(1, 8);
20. }

Bảng bên dưới thể hiện quá trình duyệt từ đỉnh 1 đến đỉnh 8.

| **Đỉnh đang xét** | **Các đỉnh kề** | **Open** | **Close** |
| --- | --- | --- | --- |
|  |  | {1} | {} |
| 1 | {2; 3; 4} | {2, 3, 4} | {1} |
| 2 | {1; 5; 6} | {3; 4; 5; 6} | {1; 2} |
| 3 | {1; 4; 5; 7; 8; 9} | {4; 5; 6; 4; 5; 7; 8; 9} | {1; 2; 3} |
| 4 | {1; 3} | {5; 6; 4; 5; 7; 8; 9} | {1; 2; 3; 4} |
| 5 | {2; 3} | {6; 4; 5; 7; 8; 9} | {1; 2; 3; 4; 5} |
| 6 | {2} | {4; 5; 7; 8; 9} | {1; 2; 3; 4; 5; 6} |
| 7 | {3} | {8; 9} | {1; 2; 3; 4; 5; 6; 7} |
| 8 | {3} | {9} | {1; 2; 3; 4; 5; 6; 7; 8} |

1. Ưu điểm

* Xét duyệt tất cả các đỉnh để trả về kết quả.
* Nếu số đỉnh là hữu hạn, thuật toán chắc chắn tìm ra kết quả.

1. Khuyết điểm

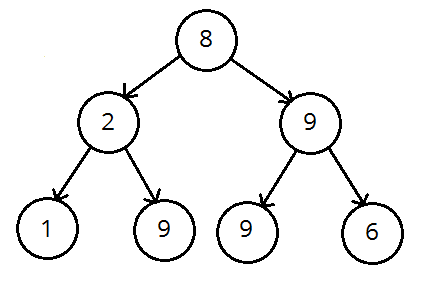
* Mang tính chất vét cạn, không nên áp dụng nếu duyệt số đỉnh quá lớn.
* Mang tính chất mù quáng, duyệt tất cả đỉnh, không chú ý đến thông tin trong các đỉnh để duyệt hiệu quả, dẫn đến duyệt qua các đỉnh không cần thiết.

##### Cây nhị phân

1. Giới thiệu

Bài viết này đề cập sơ lược đến cây nhị phân, giúp các bạn dễ dàng tiếp cận hơn khi nói về những kiểu cây nhị phân mở rộng hơn ở những bài kế tiếp.

1. Cây nhị phân

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_3/407/ss_1.png)

Cây nhị phân là tập hợp các nút (node) chứa giá trị liên kết với nhau theo quan hệ cha - con (lần lượt theo chiều mũi tên như hình vẽ) sao cho mỗi nút không có quá 2 nút con.

1. Cây con

Mỗi nút trong cây cùng với những node phía dưới nó tạo thành một cây con.

1. Nút gốc

Nút gốc là nút không là con của bất kỳ của nút nào, là nút bắt đầu của cây nhị phân, từ đó cây nhị phân mở rộng ra.

1. Nút trong

Nút trong là nút có con, bất kể 1 hoặc 2 nút con.

1. Nút lá

Nút lá là nút không có nút con nào.

1. Triển khai
2. Định nghĩa cấu trúc nút

Trong bài viết này tôi xác định giá trị chứa trong nút có kiểu int. Ngoài ra còn có hai biến con trỏ trỏ đến địa chỉ của hai nút con của mỗi nút.

1. // ...
2. struct node
3. {
4. int info;
5. struct node \*pLeft;
6. struct node \*pRight;
7. }
8. typedef struct node NODE;
9. Định nghĩa cây nhị phân

Tôi sẽ quản lý cây nhị phân từ một nút gốc ban đầu (nút không là con của nút nào khác) rồi dần duyệt xuống. Các nút sẽ được thể hiện dưới dạng con trỏ được cấp phát động để có thể trình bày các dòng code dễ dàng hơn cũng như thuận tiện tạo nút mới.

1. // ...
2. typedef NODE\* TREE;
3. Khởi tạo cây nhị phân rỗng

Khi cây chưa có nút nào, hay còn gọi là cây rỗng, nút gốc ban đầu cần được trỏ đến một địa chỉ xác định nào đó có thể có thể kiểm soát được. Như vậy chúng ta mới có thể xác định được cây nhị phân hiện tại có rỗng hay chưa.

1. // ...
2. void init(TREE &t)
3. {
4. t = NULL;
5. }
6. Khởi tạo một nút với giá trị cho trước

Cấp phát động một nút và gán địa chỉ 2 nút con của nút đó có giá trị NULL mặc định nút đó là nút lá.

1. NODE\* getNode(int x)
2. {
3. NODE\*p = new NODE;
5. // truong hop cap phat cho p that bai, tra ve gia tri NULL
6. if (p == NULL)
7. return NULL;
9. // gan gia tri chua trong nut
10. p->info = x;
12. // gan dia chi hai nut con cua nut p tam thoi la NULL
13. p->pLeft = NULL;
14. p->pRight = NULL;
16. return p;
17. }
18. Thêm nút vào cây nhị phân

Tuỳ vào ý đồ xây dựng và sắp xếp các nút trên cây nhị phân của mỗi người, chúng ta sẽ có những cách thêm nút phù hợp tương ứng. Xác định rõ tính chất của kiểu cây nhị phân và yêu cầu của bản thân để có cách phù hợp.

1. Duyệt cây nhị phân theo thứ tự

Tuỳ vào mục đích sử dụng hoặc sở thích, chúng ta có các phương pháp duyệt cây với thứ tự khác nhau:

* Phương pháp LNR (Left Node Right).
* Phương pháp LRN (Left Right Node).
* Phương pháp NLR (Node Left Right).
* Phương pháp NRL (Node Right Left).
* Phương pháp RNL (Right Node Left).
* Phương pháp RLN (Right Left Node).

Tuỳ vào các mục đích duyệt cây để thực hiện công việc gì, chúng ta sẽ có hàm duyệt cây tương ứng. Dưới đây, tôi sẽ trình bày hàm duyệt cây để lần lượt in ra các giá trị của mỗi nút theo đệ quy bằng ngôn ngữ C++.

Duyệt theo NLR (Node Left Right)

1. // ...
2. void show(TREE t)
3. {
4. // kiem tra cay nhi phan co rong khong
5. // hoac khi de quy den nut la ket thuc de quy
6. if (t == NULL)
7. return;
9. printf("%d ", t->info);
10. show(t->pLeft);
11. show(t->pRight);
12. }

Duyệt theo LNR (Left Node Right)

1. // ...
2. void show(TREE t)
3. {
4. // kiem tra cay nhi phan co rong khong
5. // hoac khi de quy den nut la ket thuc de quy
6. if (t == NULL)
7. return;
9. show(t->pLeft);
10. printf("%d ", t->info);
11. show(t->pRight);
12. }

Tương tự thay đổi thứ tự cho những cách duyệt còn lại.

### Thuật toán tìm đường A\*

A\* là giải thuật tìm kiếm trong đồ thị, tìm đường đi từ một từ một đỉnh hiện tại đến đỉnh đích có sử dụng hàm để ước lượng khoảng cách hay còn gọi là hàm Heuristic. Vậy Heuristic là gì? Heuristic là phương pháp giải quyết vấn đề dựa trên phỏng đoán, ước chừng, kinh nghiệm, trực giác để tìm ra giải pháp gần như là tốt nhất, nhanh chóng, dễ dàng.Hàm Heuristic là gì? Hàm Hueristic là hàm ứng với mỗi trạng thái hay mỗi sự lựa chọn một giá trị ý nghĩa đối với vấn đề dựa vào giá trị hàm này ta lựa chọn hành động.

Từ trạng thái hiện tại A\* xây dựng tất cả các đường đi có thể đi dùng hàm ước lược khoảng cách ( hàm Heuristic) để đánh giá đường đi tốt nhất có thể đi.Tùy theo mỗi dạng bài khác nhau mà hàm Heuristic sẽ được đánh giá khác nhau. A\* luôn tìm được đường đi ngắn nhất nếu tồn tại đường đi như thế.

A\* lưu giữ một tập các đường đi qua đồ thị, từ đỉnh bắt đầu đến đỉnh kết thúc. Tập các đỉnh có thể đi tiếp được lưu trong tập Open. Thứ tự ưu tiên cho một đường đi đươc quyết định bởi hàm Heuristic được đánh giá f(x)=g(x)+h(x). Trong đó, g(x) là chi chi phí của đường đi từ điểm xuất phát cho đến thời điểm hiện tại, h(x) là hàm ước lượng chi phí từ đỉnh hiện tại đến đỉnh đích f(x) thường có giá trị càng thấp thì độ ưu tiên càng cao.

1. Tiền đề bài viết

Trong quá trình học môn trí tuệ nhân tạo tôi được học qua thuật giải A\* và nhận thấy nó rất có ích trong việc áp dụng nó để tìm đường đi ngắn nhất giữa hai điểm hay các bài toán tìm kiếm không gian trạng thái trong đồ thị. Vì vậy qua bài viết này tôi muốn chia sẽ với các bạn về thuật giải A\*.

1. Thuật giải A\*

Gọi Open: tập các trạng thái đã được sinh ra nhưng chưa được xét đến.

Close: tập các trạng thái đã được xét đến.

Cost(p,q): là khoảng cách giữa p,q.

g(p): khoảng cách từ trạng thái đầu đến trạng thái hiện tại p.

h(p): giá trị được lượng giá từ trạng thái hiện tại đến trạng thái đích.

f(p)=g(p)+h(p).

Bước 1:

Open:={s}

Close:={};

Bước 2: while (Open !={} )

2.1 Chọn trạng thái ( đỉnh ) tốt nhất p trong Open ( xóa p khỏi Open).

2.2 Nếu p là trạng thái kết thúc thì thoát.

2.3 Chuyển p qua Close và tạo ra các trạng thái kế tiếp q sau p

2.3.1 Nếu q đã có trong Open

Nếu (g(q) > g(p) +cost(p,q))

g(q) = g(p) +cost(p,q)

f(q) = g(q) + h(q);

prev(q) = p // đỉnh cha của q là p

2.3.2 Nếu q chưa có trong Open

g(q) = g(p) + cost(p,q);

f(q) = g(q) + h(q);

prev(q) = p;

Thêm q vào Open.

2.3.3 Nếu q có trong Close

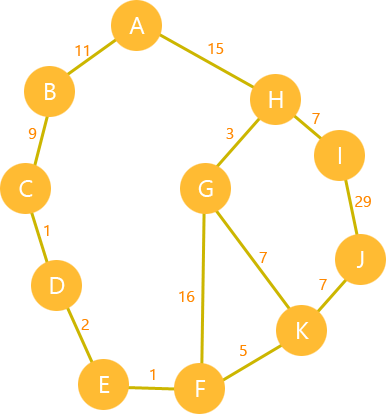
If (g(q)>g(p)+cost(p,q))

Bỏ q khỏi Close;

Thêm q vào Open.

Bước 3: Không tìm được.

1. Mô phỏng trên đồ thị

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_3/262/ss_1.png)

h(A) = 60 / h(B) = 53 / h(C) = 36 / h(D) = 35 / h(E) = 35 / h(F) = 19 / h(G) = 16 / h(H) = 38 / h(I) = 23 / h(J) = 0 / h(K) = 7

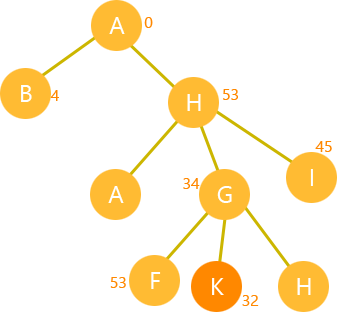
Đỉnh bắt đầu A.

Đỉnh kết thúc  K.

Ước lượng khoảng cách từ đỉnh hiện tại cho đến đỉnh kết thúc f(x)=g(x)+h(x) trong đó g là khoảng cách ngắn nhất từ đỉnh hiện tại đến đích. VD f(A) = 0 + 60.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Bước** | **P** | **Các đỉnh nối với P** | **Open** | **Close** |
| 0 |  |  | A60 |  |
| 1 | A | B, H | B64, H53 | A |
| 2 | H | G, I, A | B64, G34, I45 | A, H |
| 3 | G | H, K, F | B64, I45, K32, F53 | A, H, G |
| 4 | K | G, F, J | B64, J32, F49, I45 | A, H, G |
| 5 | K (dừng) |  |  |  |

Cây tìm kiếm ứng với đồ thị trên.

[](http://www.stdio.vn/statics/external_data/files/pages/articles/2015/quarter_3/262/ss_2.png)

Từ đó mô phỏng thuật toán trên C++ dựa trên đồ thị ở trên.

1. Hiện thực giải thuật A\*

Cách lưu các giá trị trên cây đồ thị

Dùng 2 file **Input1.txt** và**Input2.txt** để lưu  các trọng số trên cây đồ thị.

File **Input1.txt**lưu giá trị h của mỗi node mà đề bài cho còn file **Input2.txt** lưu dưới dạng ma trận lưu khoảng cách giữa 2 điểm nếu giữa 2 điểm không có đoạn nối đánh 0 (tức khoảng cách giữa hai đỉnh này bằng không hoặc không có đoạn nối 2 đỉnh này).

Nội dung file **Input1.txt**như sau:

1. 11
2. 60 53 36 35 35 19 26 38 23 0 7

Trong đó:

* 11 là số đỉnh.
* Mảng ở dưới là lưu các giá trị h của mỗi đỉnh theo thứ tự.

Nội dung file **Input2.txt** lưu:

1. 11
2. 0 11 0 0 0 0 0 15 0 0 0
3. 11 0 9 0 0 0 0 0 0 0 0
4. 0 9 0 1 0 0 0 0 0 0 0
5. 0 0 1 0 2 0 0 0 0 0 0
6. 0 0 0 2 0 11 0 0 0 0 0
7. 0 0 0 0 11 0 16 0 0 0 5
8. 0 0 0 0 0 16 0 3 0 0 7
9. 15 0 0 0 0 0 3 0 7 0 0
10. 0 0 0 0 0 0 0 7 0 29 0
11. 0 0 0 0 0 0 0 0 29 0 7
12. 0 0 0 0 0 5 7 0 0 7 0

Trong đó:

* 11 là số đỉnh
* Ma trận kề ở dưới lưu mỗi liên hệ giữa 2 đỉnh  và độ dài 2 đỉnh đó trong đồ thị theo thứ tự của các đỉnh.

Sau đó tạo 1 file main.cpp  lưu đoạn code dưới này  và chạy chương trình. Chương trình cho kết quả thứ tự các node đi qua từ điểm bắt đầu đến điểm kết thúc.

1. #include<fstream>
2. #include<iostream>
3. using namespace std;
4. struct Node
5. {
6. int stt;// so thu tu
7. int g;// khoang cach tu dinh ban dau den dinh hien ta
8. int f; // f=h+g;
9. int h;// duong di ngan nhat
10. int color;// danh dau dinh di qua
11. int dad;// dinh cha
12. };
13. int a[100][100];
14. Node p[100];
15. Node Open[100];
16. Node Close[100];
18. void ReadfileInput1(int \*b, int &n)
19. {
20. fstream myfile("Input1.txt");
21. if (!myfile.is\_open())
22. {
23. cout << "Khong the mo duoc file" << endl;
25. }
26. else
27. {
28. myfile >> n;
29. for (int i = 0; i < n; i++)
30. {
31. myfile >> b[i];

34. }
35. }
36. }
38. void ReadfileInput2(int a[100][100], int &n, int &start, int &finsh)
39. {
41. fstream shichiki("Input2.txt");
42. if (!shichiki.is\_open())
43. {
44. cout << "Khong the mo duoc file !";
45. }
46. else
47. {
48. shichiki >> n >> start >> finsh;
50. for (int i = 0; i < n; i++)
51. {
52. for (int j = 0; j < n; j++)
53. shichiki >> a[i][j];
54. }
55. }
56. shichiki.close();
58. }
59. void printfmaxtric(int a[100][100], int n)
60. {
61. for (int i = 0; i < n; i++)
62. {
63. for (int j = 0; j < n; j++)
64. {
65. cout << a[i][j] << "\t";
66. }
67. cout << "\n";
68. }
70. }
72. int Count(int n, Node \*Open)
73. {
74. int count = 0;
75. for (int i = 0; i < n; i++)
76. {
77. if (Open[i].color == 1)
78. count = count + 1;
79. }
80. return count;
81. }
82. int Find(int n, Node \*Open)
83. {
85. for (int i = 0; i < n; i++)
86. if (Open[i].color == 1)
87. return i;
88. }
89. int Findmin(int n, Node \*Open)
90. {
91. int min1 = Find(n, Open);
92. int min = Open[min1].f;
93. for (int i = 0; i < n; i++)
94. {
95. if (Open[i].f < min && Open[i].color == 1)
96. {
97. min1 = i;
98. min = Open[i].f;
99. }
101. }
102. return min1;
103. }
104. void Innit(int n, int \*b)
105. {
106. for (int i = 0; i < n; i++)
107. {
108. p[i].stt = i;
109. p[i].color = 0;
110. p[i].g = b[i];
111. p[i].dad = 0;
112. p[i].f = p[i].g;
113. p[i].h = 0;
114. }
115. }
116. int Findpoint(int n, Node \*q, int o)
117. {
118. for (int i = 0; i < n; i++)
119. if (q[i].stt == o)
120. return i;
122. }
123. void astar(int a[100][100], int n, int start, int finsh, int b[])
124. {
125. int l = 0;
126. Open[l] = p[start];
127. Open[l].color = 1;
128. Open[l].f = Open[l].h + Open[l].g;
129. l++;
130. int w = 0;
131. while (Count(l, Open) != 0) // kiem tra xem tap Open co con phan tu nao khong
132. {
133. int k = Findmin(n, Open); // tim vi tri nho nhat trong Open
134. Open[k].color = 2; // cho diem tim duoc vao Close
135. Close[w] = Open[k];
136. Close[w].color = 2;
137. w++;
138. p[Findpoint(n, p, Open[k].stt)].color = 2;
139. if (Findpoint(n, p, Open[k].stt) == finsh)
140. {
141. cout << "Duong di qua la" << endl;
142. cout << finsh << "\t";
143. int y = Findpoint(w, Close, finsh);
144. int u = Close[y].dad;
145. while (u != start)
146. {
147. y = Findpoint(w, Close, u);
148. u = Close[y].dad;
149. cout << u << "\t";
150. }
151. break;
152. }
153. else
154. {
155. for (int i = 0; i < n; i++)
156. {
157. if (a[Findpoint(n, p, Open[k].stt)][i] != 0 && p[i].color == 0)// neu chua co trong Open va Close
158. {
159. Open[l] = p[i];
160. Open[l].color = 1;
161. Open[l].h = a[Findpoint(n, p, Open[k].stt)][i] + Open[k].h;// tinh h khoang cach ngan nhat tu dinh bat dau den dinh hien tai
162. Open[l].f = Open[l].g + Open[l].h;
163. Open[l].dad = Findpoint(n, p, Open[k].stt);
164. p[i].color = 1;
165. l++;
166. }
167. if (a[Findpoint(n, p, Open[k].stt)][i] != 0 && p[i].color == 1)// neu dinh da co trong open
168. {
169. int h = Findpoint(l, Open, p[i].stt);
170. Node tam = p[i];
171. tam.color = 1;
172. tam.h = a[Findpoint(n, p, Open[k].stt)][i] + Open[k].h;
173. tam.dad = k;
174. tam.f = tam.h + tam.g;
175. if (tam.f < Open[h].f)//neu f trang thai hien tai be hon trang thai cap nhat truoc do
176. Open[h] = tam;
177. }
178. if (a[Findpoint(n, p, Open[k].stt)][i] != 0 && p[i].color == 2)// neu dinh da co trong Close
179. {
180. int h = Findpoint(l, Close, p[i].stt);
181. Node tam = p[i];
182. tam.color = 1;
183. tam.h = a[Findpoint(n, p, Open[k].stt)][i] + Open[k].h;
184. tam.dad = k;
185. tam.f = tam.h + tam.g;
186. if (tam.f < Close[h].f)//neu f trang thai hien tai be hon trang thai truoc do
187. {
188. Open[l] = tam;// them vao Open
189. Close[h].color = 1;//danh dau dinh do thuoc Open
190. l++;
191. }
192. }
193. }
194. }
195. }
196. }
198. int main()
199. {
200. int n;
201. int start;
202. int finsh;
203. int b[100];
205. ReadfileInput2(a, n, start, finsh);
206. ReadfileInput1(b, n);
208. Innit(n, b);
209. cout << "Dinh bat dau" << endl;
210. cout << start << endl;
211. cout << "Dinh ket thuc" << endl;
212. cout << finsh << endl;
214. astar(a, n, start, finsh, b);
215. return 0;
216. }
217. Nhận xét
218. Ưu điểm

Một thuật giải linh động, tổng quát, trong đó hàm chứa cả tìm kiếm chiều sâu, tìm kiếm chiều rộng và những nguyên lý Heuristic khác.Nhanh chóng tìm đến lời giải với sự định hướng của hàm Heuristic. Chính vì thế mà người ta thường nói, A\* chính là thuật giải tiêu biểu cho Heuristic.

1. Nhược điểm

A\* rất linh động nhưng vẫn gặp một khuyết điểm cơ bản – giống như chiến lược tìm kiếm chiều rộng – đó là tốn khá nhiều bộ nhớ để lưu lại những trạng thái đã đi qua.

### Thuật toán nén

1. Mã hóa Huffman
2. Giới thiệu

Đề xuất bởi tiến sỹ David A.Huffman năm 1952 trong “phương pháp xây dựng mã hóa tối thiểu”, ứng dụng nhiều trong truyền dữ liệu.

Chúng ta sẽ ứng dụng trên văn bản.

1. Cơ bản về giải thuật

Sau đây là trình tự thuật toán:

* Quét dữ liệu để nén và kiểm tra sự xuất hiện các kí tự.
* Sắp xếp các kí tự dựa trên số lần xuất hiện trong văn bản
* Xây dựng cây mã hóa Huffman dựa trên danh sách đã sắp xếp ở trên
* Duyệt cây để xác định tất cả các từ mã hóa
* Quét văn bản lần nữa và xuất file dùng mã hóa Huffman.

### Triển khai Ma trận với STL

Several algorithms in Chapter 10 use two-dimensional arrays, which are popularly known

as matrices. The C++ library does not provide a matrix class. However, a reason

able matrix class can quickly be written. The basic idea is to use a vector of vectors.

Doing this requires additional knowledge of operator overloading. For the matrix, we

define operator[], namely, the array-indexing operator. The matrix class is given in

Figure 1.26.

##### Thành viên dữ liệu, hàm dựng và Accessors

The matrix is represented by an array data member that is declared to be a vector of

vector<Object>. The constructor first constructs array as having rows entries each of type

vector<Object> that is constructed with the zero-parameter constructor. Thus, we have rows

zero-length vectors of Object.

The body of the constructor is then entered, and each row is resized to have cols

columns. Thus the constructor terminates with what appears to be a two-dimensional

array. The numrows and numcols accessors are then easily implemented, as shown.

#ifndef MATRIX\_H

#define MATRIX\_H

#include <vector>

using namespace std;

template <typename Object>

class matrix{

public:

matrix( int rows, int cols ) : array( rows ){

for( auto & thisRow : array )

thisRow.resize( cols );

}

matrix( vector<vector<Object>> v ) : array{ v }{ }

matrix( vector<vector<Object>> && v ) : array{ std::move( v ) }{ }

const vector<Object> & operator[]( int row ) const{ return array[ row ]; }

vector<Object> & operator[]( int row ){ return array[ row ]; }

int numrows( ) const{ return array.size( ); }

int numcols( ) const{ return numrows( ) ? array[ 0 ].size( ) : 0; }

private:

vector<vector<Object>> array;

};

#endif

##### Toán tử [ ]

The idea of operator[] is that if we have a matrix m, then m[i] should return a vector

corresponding to row i of matrix m. If this is done, then m[i][j] will give the entry in

position j for vector m[i], using the normal vector indexing operator. Thus, the matrix

operator[] returns a vector<Object> rather than an Object.

We now know that operator[] should return an entity of type vector<Object>. Should

we use return-by-value, return-by-reference, or return-by-constant-reference? Immediately

we eliminate return-by-value, because the returned entity is large but guaranteed to exist

after the call. Thus, we are down to return-by-reference or return-by-constant-reference.

Consider the following method (ignore the possibility of aliasing or incompatible sizes,

neither of which affects the algorithm):

void copy( const matrix<int> & from, matrix<int> & to ){

for( int i = 0; i < to.numrows( ); ++i )

to[ i ] = from[ i ];

}

In the copy function, we attempt to copy each row in matrix from into the corresponding

row in matrix to. Clearly, if operator[] returns a constant reference, then to[i] cannot

appear on the left side of the assignment statement. Thus, it appears that operator[] should

return a reference. However, if we did that, then an expression such as from[i]=to[i] would

compile, since from[i] would not be a constant vector, even though from was a constant

matrix. That cannot be allowed in a good design.

So what we really need is for operator[] to return a constant reference for from, but

a plain reference for to. In other words, we need two versions of operator[], which differ

only in their return types. That is not allowed. However, there is a loophole: Since member

function const-ness (i.e., whether a function is an accessor or a mutator) is part of the

signature, we can have the accessor version of operator[] return a constant reference, and

have the mutator version return the simple reference. Then, all is well. This is shown in

Figure 1.26.

##### Big-Five

These are all taken care of automatically, because the vector has taken care of it. Therefore,

this is all the code needed for a fully functioning matrix class.

### Thư viện template chuẩn STL

##### Tổng quan về C++ STL

Bạn có thể tham khảo lý thuyết về [**C++ Template tại đây**](#_Template_trong_C++)

C++ STL (Thư viện template chuẩn) là một bộ các lớp template vô cùng mạnh mẽ cung cấp các lớp và hàm được template hóa, triển khai các thuật toán và cấu trúc dữ liệu như vector, link-list, queues và stack.

Điểm thuận lợi là bạn không cần phải viết và debug, và STL đã làm tốt công việc của nó để cung cấp một phiên bản hiệu quả, hợp lý các lớp này. Mặc dù STL khá là phức tạp khi được template hóa.

Các thành phần cốt lõi của C++ STL bao gồm:

|  |  |
| --- | --- |
| **Thành phần** | **Mô tả** |
| Containers | Là bộ chứa (hay mảng) các đối tượng cụ thể. Có nhiều loại khác nhau như deque, list, vector, map… |
| Algorithms | Giải thuât triển khai trên bộ chứa đối tượng. Cho phép khởi tạo, sắp xếp, tìm kiếm và biến đổi |
| Iterators | Bộ lặp dể duyệt qua các phần tử đối tượng. Bộ này chứa các thành phần con của bộ chứa.. |

Cả ba thành phần trên được định nghĩa rất nhiều hàm giúp chúng ta thao tác trên mảng các đối tượng dễ dàng.

1. STL container

Chia làm nhiều lớp chứa dùng trong nhiều tình huống khác nhau. Nói chung, lớp chứa rơi vào 3 nhóm sau:

* Lớp chứa tuần tự
* Lớp chứa liên kết
* Lớp chứa nối tiếp

1. Lớp chứa tuần tự

Gồm 3 lớp : vector, deque và list. Tương ứng định nghĩa trong 3 thư viện <vector>, <deque>, <list>

Có vai trò duy trì trật tự của nhiều phần tử trong lớp chứa. Một trong đặc tính của lớp chứa tuần tự là có thể chèn một phần tử vào vị trí bất kì.

Công dụng phổ biến nhất là biểu diễn mảng: nếu bạn chèn 4 phần tử vào mảng, các phần tử sẽ nằm đúng vị trí bạn muốn.

* Vector: trong STL là một mảng động có khả năng tự mở rộng vùng nhớ cần thiết để chứa các phần tử. Lớp Vector cho phép truy xuất ngẫu nhiên phần tử qua toán tử [] hay phương thức .at(), và chèn, xóa phần tử cuối khá nhanh

Đoạn code minh họa chèn 6 phần tử cuối Vector:

vector<int> vect;

for (int nCount=0; nCount < 6; nCount++)

vect.push\_back(nCount); // chèn vào cuối mảng

* Deque: là lớp hàng đợi kép, là một mảng động có thể giãn theo cả 2 đầu.

deque<int> deq;

for (int nCount=0; nCount < 3; nCount++) {

deq.push\_back(nCount); // chèn vào cuối mảng

deq.push\_front(10 - nCount); // chèn vào đầu mảng

}

* List : là lớp chứa kiểu danh sách liên kết kép, mỗi phần tử chứa con trỏ trỏ tới phần tử kế tiếp và con trỏ trỏ tới phần tử phía sau.

Lớp List chỉ cho truy xuất phần tử đầu và cuối của danh sách – không cho truy xuất ngẫu nhiên.

Nếu bạn muốn tìm một phần tử ở giữa danh sách, bạn phải duyệt danh sách. List cho phép bạn chèn phần tử khá nhanh nếu biết vị trí.  
Chúng ta sẽ dùng Iterator để duyệt danh sách (sẽ đề cập phần sau)

Mặc dù lớp STL string và wstring không được chuẩn hóa trong lớp chứa tuần tự, nhưng bạn có thể dùng vector thay thế với kiểu dữ liệu <char> hay <wchar>.

1. Lớp chứa liên kết

Các lớp này tự động sắp xếp dữ liệu nhập khi dữ liệu được chèn vào bộ chứa lớp. Mặc định, lớp chứa liên kết so sánh các phần tử qua toán tử (<).

* **set** là bộ chứa để lưu trữ các phần tử độc nhất, không cho phép đúp phần tử. Các phần tử được sắp xếp theo giá trị.
* **multiset** là một set cho phép đúp số phần tử.
* **map** (hay gọi là mảng liên kết) là một set có mỗi phần tử là một cặp **key** - giá trị. Key dùng để sắp xếp và đánh chỉ mục dữ liệu, và **key** phải là duy nhất.
* **multimap** (hay gọi là từ điển) là một bản đồ cho phép đúp các **key**. Từ điển cuộc sống là một multimap: **key** là thế giới, giá trị là ý nghĩa của thế giới.

Tất cả key được sắp xếp tằng dần, và bạn có thể tra cứu giá trị bằng **key**. Một vài từ có nhiều ý nghĩa khác nhau, đó là lý do tại sao từ điển phải là dạng multimap hơn là map.

1. Lớp chứa tiếp nối

Là kiểu đặc biệt dành cho đối tượng người dùng cụ thể, ban có thể chọn lớp chứa nào cần làm việc.

* Stack : ngăn xếp là lớp chứa vận hành kiểu **LIFO** (Last In, Fisrt Out), thao tác rất đơn giản chỉ có chèn (pushed) và xóa (poped) từ phần tử cuối. Stack dùng **deque** như lớp chứa tuần tự mặc định (mặc dù vector lại có vẻ phù hợp hơn), nhưng chúng ta có thể dùng **Vector** hay **List** cũng được.
* Queue: hàng đợi là lớp chứa kiểu **FIFO** (First In, First Out), các phần tử có thể chèn (pushed) vào cuối và xóa (poped) từ phía trước.  
   Queue dùng **deque** làm lớp chứa mặc định, nhưng ta có thể dùng **List**.
* Priority queue: hàng đợi ưu tiên có phần tử luôn được sắp xếp qua toán tử (<). Khi phần tử được đẩy vào (pushed), nó sẽ được sắp xếp ngay trong hàng đợi.  
  Xóa phần tử ở phía trước hàng đợi sẽ trả quyền ưu tiên lại cho phần tử còn lại.

##### Iterator

Khi thao tác với các phần tử trên List, chúng ta phải nhận biết vị trí. Vị trí phần tử trong các container của STL đại diện bởi kiểu dữ liệu lồng bên trong container là **iterator**.

Ví dụ: list<string>::iterator; vector<int>::iterator

Có 3 vấn đề cần quan tâm:

* Cách lấy iterator
* Thao tác trên iterator
* Các phương thức nào trên lớp trừu tượng nhận iterator làm tham số

1. Sử dụng Iterator

Trên hầu hết STL container đều hỗ trợ 2 phương thức lấy iterator:

* iterator begin( ): trả về iterator đại diện cho phần tử đầu tiên trong container.
* iterator end( ): trả về iterator được đánh dầu là kết thúc trong container (nằm sau phần tử cuối cùng).

Đây là đoạn code dùng vòng lặp in tất cả phần tử vector trong C++11:

for( int i = 0; i != v.size( ); ++i )

cout << v[ i ] << endl;

Và nếu sử dụng iterator thì là dạng tương tự như:

for( vector<int>::iterator itr = v.begin( ); itr != v.end( ); itr.??? )

cout << itr.??? << endl;

Vậy chúng ta dùng phương thức gì trên iterator để có thể duyệt như cách tăng giá trị biến I trong vòng lặp bên trên?

1. Thao tác trên Iterator

Iterator được hỗ trợ quá tải toán tử so sánh !=, ==, và có hàm dựng sao chép, toán tử gán. Do đó, sẽ có những phương thức sử dụng toán tử đa năng của iterator, chúng ta có thể liệt kệ:

* itr++ và ++itr: chuyển tới iterator đại diện phần tử kế tiếp trong container.
* \*itr: trả về tham chiếu tới phần tử mà iterator đại diện. Tham chiếu này có thể hoặc không thay đổi giá trị được (bàn sau).
* itr1==itr2: so sánh xem 2 iterator có cùng tham chiếu tới cùng 1 phần tử không.
* itr1!=itr2: trả về true nếu 2 iterator tham chiếu tới 2 phần tử khác nhau, hoặc false nếu ngược lại.

Vòng lặp trong đoạn code trên sẽ được tối ưu như sau:

for( vector<int>::iterator itr = v.begin( ); itr != v.end( ); ++itr )

cout << \*itr << endl;

Ngoài ra chúng ta có thể dùng đồng thời phép tham chiếu và vừa tăng iterator \*itr++. Dùng cấu trúc While như sau:

vector<int>::iterator itr = v.begin( );

while( itr !=v.end( ) )

cout << \*itr++ << endl;

1. Các thao tác trên container cần Iterators

Có 3 phương thức phổ biến nhất cần iterator trên List/vector là thêm, xóa hay chèn phần tử

* insert(iterator **pos**, const Object &**x**): chèn đối tượng x theo sau vị trí pos trong List
* iterator erase( iterator **pos** ): xóa phần tử tham chiếu bởi iterator pos trong List.

Thời gian truy xuất trên list là hằng, nhưng vector thì không. Hàm erase trả về iterator đại diện phần tử theo sau phần tử mới xóa.

Iterator **pos** sẽ không còn hiệu lực.

* iterator erase( iterator **start**, iterator **end** ): xóa tất cả phần tử từ iterator start cho tới, nhưng không bao gồm **end**. Để xóa toàn bộ List thì chỉ cần gọi c.erase( c.**begin**( ), c.**end**( ) ).

1. Hiệu suất trên container

Chúng ta duyệt List và dùng earase trên mỗi phần tử sau phần tử đầu tiên.

Phương thức erase trên List là tuyến tính, mỗi lần gọi mất một khoảng thời gian nhất định, nhưng trên vector sẽ tốn bình phương thời gian vì mỗi lần gọi không hiệu quả.

Chúng ta sử dụng template để da dạng hóa các container.

template <typename Container>

void removeEveryOtherItem( Container & lst ){

auto itr = lst.begin( ); // itr: Container::iterator

while( itr != lst.end( ) )

{

itr = lst.erase( itr );

if( itr != lst.end( ) )

++itr;

}

}

Dùng từ khóa **auto** để trình biên dịch tự nhận biết kiểu iterator của bất kì container nào.

Một list<int>, mất 0.039 giây cho 800,000 phần tử, và 0.073 giây cho 1,600,000 phần tử, thời gian tăng tuyến tính theo số lượng phần tử.

Tương tự, vector<int>, mất 5 phút cho 800,000 phần tử và 20 phút cho 1,600,000 phần tử, tức là thời gian tăng gấp 4, cho lương phần tử tăng gấp 2.

1. const\_iterators

\*iterator không những trả về giá trị mà còn tham chiếu tới đối tượng trong container. Lợi ích là chúng ta hoàn toàn có thể thay đổi giá trị cho từng phần tử.

Phương thức sau dùng generic code cho mọi kiểu, áp dụng trên cả vector hay list, thời gian thực thi tuyến tính:

template <typename Container, typename Object>

void change( Container & c, const Object & newValue ){

typename Container::iterator itr = c.begin( );

while( itr != c.end( ) )

\*itr++ = newValue;

}

Giả sử rằng chúng ta không muốn thay đổi dữ liệu và truyền tham chiếu hằng cho Container c. Trình biên dịch đảm bảo không cho phép các hàm thành viên mutator của c được gọi.

template <typename Container>

void print( const list<int> & lst, ostream & out = cout ){

typename Container::iterator itr = lst.begin( );

while( itr != lst.end( ) ){

out << \*itr << endl;

\*itr = 0; // Hợp lệ? !!!

++itr;

}

}

Đoạn code trên không hợp lệ và sẽ không được biên dịch.

STL cung cấp giải pháp cho vấn đề trên, bằng kiểu dữ liệu hằng iterator lồng bên trong container là **const\_iterator**.

Điểm khác biệt chủ yếu của iterator và const\_iterator là toán tử tham chiếu nội dung \*const\_iterator là kiểu r\_value.

Ngoài ra trình biên dịch bắt buộc bạn phải dùng const\_iterator để duyệt hằng container.

Tương tự như begin/end, chúng ta có phương thức hằng để lấy const\_iterator như sau:

* iterator begin( )
* const\_iterator begin( ) const
* iterator end( )
* const\_iterator end( ) const

Khi biến container là non constant, mutator begin sẽ được gọi. Ngược lại, container là hằng thì phiên bản begin const được gọi.

Nếu dùng **auto**, trình biên dịch sẽ tự nhận biết kiểu theo từng hoàn cảnh một cách chính xác hoàn toàn.

C++11 cho phép viết code làm việc dù kiểu container không có hàm thành viên begin và end.

Hàm không thành viên begin và end được định nghĩa để dùng begin(c) thay thế c.begin()tại bất kỳ nơi nào được phép

Generic code dùng begin(c) thay vì c.begin() thuận lợi hơn, nhờ làm việc được với container có hay không có thành viên begin và end.

template <typename Container>

void print( const Container & c, ostream & out = cout ) {

if( c.empty( ) )

out << "(empty)";

else

{

auto itr = begin( c ); // itr is a Container::const\_iterator

out << "[ " << \*itr++; // Print first item

while( itr != end( c ) )

out << ", " << \*itr++;

out << " ]" << endl;

}

}

Ngoài ra, hàm tự do begin và end trong C++11 có sử dụng **auto** và **decltype**, nên chúng tự động xác định được kiểu trả về thông qua kiểu tham số truyền vào.

template<typename Container>

auto begin( Container & c ) -> decltype( c.begin( ) )

{

return c.begin( );

}

template<typename Container>

auto begin( const Container & c ) -> decltype( c.begin( ) )

{

return c.begin( );

}

1. Toán tử , hàm với iterator

Một iterator được ảo hóa hoàn hảo như một con trỏ, trỏ tới các phần tử của lớp chứa, với một bộ các quá tải toán tử định nghĩa rất phong phú:

* Toán tử truy xuất nội dung \* : Iterator trả về phần tử hiện tại mà Iterator đang trỏ tới.
* Toán tử ++ và -- : Chuyển Iterator tới phần tử kế tiếp hay trở về phần tử trước.
* Toán tử == và != : so sánh để xác định 2 Iterator có trỏ tới cùng phần tử không. Để so sánh giá trị, bạn dùng toán tử nội dung trước khi so sánh.
* Toán tử = : Gán Iterator tới vị trí mới (thường là phần tử đầu hay cuối của lớp chứa). Để gán giá trị, bạn cũng phải truy xuất nội dung trước khi gán.
* Hàm **std:advance**(<iterator>, <step>): nhảy Iterator lên/xuống tương ứng với giá trị dương/âm của **step**

1. Hàm thành viên collection

Mỗi lớp chứa gồm 4 hàm thành viên cơ bản để dùng với toán tử gán (=) là:

* begin() : trả về iterator đại diện cho phần tử đầu tiên của lớp chứa.
* end() : trả về iterator đại diện phần tử cuối của lớp chứa.
* cbegin() : trả về hằng iterator đại diện phần tử đầu tiên của lớp chứa.
* cend() : trả về hằng iterator đại diện phần tử cuối của lớp chứa.

Các phương thức trên giúp duyệt danh sách dễ dàng: lặp qua các phần tử cho tới khi gặp end().

Cuối cùng là tất cả lớp chứa đều có 2 loại iterator:

* container::iterator là iterator dùng dể truy xuất và có thể thay đổi giá trị phần tử.
* container::const\_iterator là iterator chỉ để truy xuất nhưng không thể thay đổi giá trị (lưu ý: đây không phải là biến hằng)

1. Duyệt vector<>

Ví dụ về iterator duyệt qua một vector

int main(){

vector<int> vect;

for (int nCount=0; nCount < 6; nCount++)

vect.push\_back(nCount);

vector<int>::const\_iterator it; // khai báo iterator hằng

it = vect.begin(); // gán cho phần begin() của vector

while (it != vect.end()) // lắp tới khi gặp end()

{

cout << \*it << " "; // in ra giá trị mà nó duyệt qua

++it; // trỏ sang phần tử kế tiếp

}

cout << endl;

return 0;

}

Kết quả

0 1 2 3 4 5

1. Duyệt List<>

Ví dụ tương tự cho list:

#include <iostream>

#include <list>

using namespace std;

int main(){

list<int> li;

for (int nCount=0; nCount < 6; nCount++)

li.push\_back(nCount);

list<int>::const\_iterator it;

it = li.begin();

while (it != li.end())

{

cout << \*it << " ";

++it;

}

cout << endl;

}

Kết quả

0 1 2 3 4 5

1. Tại sao phải dùng iterator

Iterator cho phép bạn duyệt lớp chứa dễ dàng mà không cần biết cấu trúc của lớp chứa thế nào.

Toán tử truy xuất chỉ số phần tử [] chỉ được định nghĩa trên lớp chứa vector, nhưng nhiều lớp chứa khác không có (Như list, forward\_list hay các lớp chứa liên kết)

Iterator giúp bạn làm việc độc lập trên nhiều lớp chứa nếu bạn không chắc về khả năng truy xuất ngẫu nhiên hay phương thức .**size**()

Để sử dụng hiệu quả các lớp STL Algorithm, bắt buộc bạn phải dùng iterator. Iterator thể hiện sức mạnh rõ rệt khi kết hợp giữa STL algorithm và hàm thành viên của lớp chứa.

****Trong bài tiếp theo, bạn sẽ xem ví dụ về cách dùng iterator để chèn phần tử vào một danh sách (mà không dùng toán tử [] để truy xuất trực tiếp phần tử).

**Một lưu ý: iterator phải được cung cấp trong mỗi lớp cơ bản, vì iterator cần biết cách một lớp được triển khai. Do đó, iterator luôn cột chặt với lớp chứa cụ thể.**

##### STL Algorithm

1. Giới thiệu

Gồm một lượng lớn các giải thuật chung định nghĩa trong thư viện <algorithm> để làm việc với các lớp chứa. Cho phép các bạn tìm kiếm, sắp xếp, chèn, xóa và sao chép phần tử.

**Lưu ý rằng các giải thuật được triển khai như là các hàm toàn cục, có sử dụng iterator.**

Mỗi giải thuật chỉ định nghĩa một lần, nhưng lại làm việc với mọi lớp chứa có iterator (kể cả lớp chứa bạn tự định nghĩa).

Mặc dù STL Algorithm giúp bạn thao tác dễ dàng, nhanh chóng trên lớp chứa nhưng có điểm tối là : một vài lớp chứa lại không làm việc với Algorithm, gây ra lặp vô hạn hay tốn hiệu suất.

Chúng ta khai báo một list như sau:

list<int> li;

for (int nCount=0; nCount < 6; nCount++)

li.push\_back(nCount);

1. Phần tử nhỏ nhất và lớn nhất

Bạn dùng

min\_element(<iterator\_bắt\_đầu>, <iterator\_kết\_thúc>)

max\_element(<iterator\_bắt\_đầu>, <iterator\_kết\_thúc>)

list<int>::const\_iterator it; // declare an iterator

it = min\_element(li.begin(), li.end());

cout << \*it << " ";

it = max\_element(li.begin(), li.end());

cout << \*it << " ";

1. Tìm kiếm và chèn

Chúng ta sẽ dùng giải thuật

**Iterator** find(<iterator\_bắt\_đầu>,<iterator\_kết\_thúc>, <giá\_trị\_tìm>)

Để tìm ra iterator trong lớp chứa, sau đó dùng

list::insert(<iterator\_bị\_chèn>, <giá\_trị\_chèn>)

Để chèn phần tử mới phía trước.

list<int>::iterator it; // declare an iterator

it = find(li.begin(), li.end(), 3); // tìm iterator có giá trị 3 trong list

li.insert(it, 8); // chèn iterator mới có giá trị 8 phía trước

for (it = li.begin(); it != li.end(); it++) // duyệt với iterator

cout << \*it << " ";

cout << endl;

1. Sắp xếp

Chúng ta khai báo một vector như sau:

vector<int> vect;

vect.push\_back(7);

vect.push\_back(-3);

vect.push\_back(6);

vect.push\_back(2);

vect.push\_back(-5);

vect.push\_back(0);

vect.push\_back(4);

Dùng hàm

**sort**(<iterator\_bắt\_đầu>,<iterator\_kết\_thúc>,<hàm\_tính\_toán>)

* iterator\_bắt\_đầu: có thể là địa chỉ, con trỏ đầu mảng hay iterator bắt đầu lớp chứa
* iterator\_kết\_thúc: địa chỉ, con trỏ đầu mảng hay iterator kết thúc lớp chứa
* hàm\_tính\_toán: qui định hàm sắp xếp theo thuộc tính nào của đối tượng phần tử trong mảng/lớp chứa.

**Lưu ý: hai tham số đầu tiên có thể tùy biến, không nhất thiết luôn là đầu hay cuối, có thể là một đoạn trong mảng hay lớp chứa cần sắp xếp.**

ĐỂ SẮP XẾP TĂNG DẦN

**sort**(vect.begin(), vect.end()); // sắp xếp

vector<int>::const\_iterator it; // khai báo iterator chỉ đọc

for (it = vect.begin(); it != vect.end(); it++) // duyệt

cout << \*it << " ";

cout << endl;

DÙNG THAM SỐ THỨ 3

Tham số này qui định hai tiêu chí sau:

* Tăng hay giảm dần
* Sắp xếp theo thuộc tính nào (của đối tượng)

Hàm tính toán phải có dạng như sau:

**bool** <tên\_hàm>(**const** <kiểu> &<đối\_tượng\_1>, **const** <kiểu> &<đối\_tượng\_2>){

**return** <đối\_tượng\_1>.<thuộc\_tính> <toán\_tử\_quan\_hệ> <đối\_tượng\_1>.<thuộc\_tính>;

}

* <toán\_tử\_quan\_hệ> : hàm sort() dựa vào biểu thức quan hệ trong hàm tính toán để sắp xếp (> giảm dần, < tăng dần) theo thuộc tính nào của đối tượng.

Ví dụ

Khai báo cấu trúc Person:

struct Person{

string name;

int age;

};

Khai báo hàm tính toán sắp tăng dần cho hai thuộc tính

// hàm sắp theo name

bool sortByName(const Person &p1, const Person &p2) { return p1.name < p2.name; }

// hàm sắp theo age

bool sortByAge(const Person &p1, const Person &p2) { return p1.age < p2.age; }

Ngoài ra bạn có thể quá tải hàm tính toán cho các thuộc tính tăng hay giảm dần. Sau đó truyền đúng tên hàm cần vào sort()

Sắp xếp theo name:

// Sap xep theo name

sort(people.begin(), people.end(), **sortByName**);

for (Person &n : people)

cout << n.name << " ";

Sắp xếp theo age:

// Sap xep theo age

sort(people.begin(), people.end(), **sortByAge**);

for (Person &n : people)

cout << n.age << " ";



**Lưu ý hàm sort() sẽ không làm việc với lớp chứa list – lớp này đã có hàm thành viên .sort() , làm việc hiệu quả hơn so với phiên bản chung của algorithm.**

1. Đảo ngược phần tử

Cú pháp

**reverse**(<iterator\_bắt\_đầu>,<iterator\_kết\_thúc>)

Để đảo ngược vị trí các phần tử trong collection

**reverse**(vect.begin(), vect.end()); // đảo vị trí đối xứng toàn danh sách

for (it = vect.begin(); it != vect.end(); it++) // duyệt

cout << \*it << " ";

cout << endl;

1. Xử lí tự động collection bằng for\_each

Mục đích của chúng ta là xử lí cùng một phương thức trên hàng loạt phần tử trong collection

Cú pháp:

**For\_each**(<iterator\_bắt\_đầu>,<iterator\_kết\_thúc>, <hàm\_1\_tham\_số> )

Tham số thứ 3 sẽ là con trỏ hàm hay Functor nhận 1 tham số cho phần tử trên collection. Hàm phải có dạng như sau:

void fun(const Type &a)

void fun(Type a)

Kiểu **Type** phải khớp với kiểu dữ liệu của **iterator**. Nếu con trỏ hàm có trả về thì sẽ bị bỏ qua.

Lưu ý: Nếu tham số thứ 3 là Functor thì không được dùng hoặc phải **delete** hàm dựng và phép gán Movie semantic

Ví dụ:

struct Sum{

Sum(){sum = 0;}

void operator()(int n) { sum += n; }

int sum;

};

int main(){

std::vector<int> nums{4,5,3,9,16,268};

// gọi Sum::operator() cho mỗi phần tử

Sum s = std::for\_each(nums.begin(), nums.end(), Sum());

// xuất từng phần tử

for (auto const &n : nums) {

std::cout << ' ' << n;

}

std::cout << '\n';

std::cout << "sum: " << s.sum << '\n';

return 0;

}

Kết quả:

4 5 3 9 16 268

sum: 305

##### Std::vector

Tương tự như std::array, thư viện chuẩn C++ cung cấp một template khác là std::vector, để làm việc với mảng động an toàn và thuận lợi hơn.

Std::vector có các tính năng gần giống với mảng tĩnh truyền thống.

Chúng ta sẽ tìm hiểu tại sao std::vector lại trở thành công vụ hữu dụng và linh hoạt nhất trong C++.

1. Giới thiệu std::vector

Ra mắt từ C++ 03, std::vector cung cấp các chức năng mảng động để quản lý vùng nhớ được cấp phát. Nghĩa là bạn có thể tạo mảng với kích thước tùy biến trong thời gian chạy, nhưng không cần thao tác cấp phát và hủy cấp phát vùng nhớ qua **new** và **delete** như mảng động truyền thống.

Std::vector được khai báo trong thư viện **<vector>**

1. Khởi tạo mảng

Cú pháp:

Std::vector<kiểu\_dữ\_liệu> tên\_biến;

Std::vector<kiểu\_dữ\_liệu> tên\_biến = {danh sách khởi tạo}

Std::vector<kiểu\_dữ\_liệu> tên\_biến {danh sách khởi tạo chuẩn} C++11

Bạn không cần cung cấp số phần tử mảng khi khai báo, std::vector cấp phát tĩnh theo nhu cầu người dùng.

Ví dụ:

#include <vector>

std::vector<int> array; // khai báo không có số phần tử

std::vector<int> array2 = { 9, 7, 5, 3, 1 }; // dùng danh sách khởi tạo

std::vector<int> array3 { 9, 7, 5, 3, 1 }; // dùng khởi tạo chuẩn (C++11 trở đi)

1. Truy xuất phần tử

Bạn vẫn dùng toán tử [] để truy xuất, nhưng không kiểm tra ranh giới mảng. Hàm thành viên at() giúp bạn truy xuất phần tử mảng có kiểm tra ranh giới.

array[2] = 2;

array.at(3) = 3;

Trên C++11, bạn có thể gán danh sách khởi tạo cho std::vector

array = { 0, 1, 2, 3, 4 }; // okay, array size is now 5

array = { 9, 8, 7 }; // okay, array size is now 3

1. Tự động hủy vùng nhớ

Khi một biến vector đi ra ngoài phạm vi, nó tự động hủy vùng nhớ cấp phát mà nó quản lý (nếu cần). Không những thế, nó còn ngăn bộ nhớ bị thiếu hụt.

Xem đoạn code sau:

void doSomething(bool earlyExit){

int \*array = new int[5] { 9, 7, 5, 3, 1 };

if (earlyExit)

return 0;

// do stuff here

delete[] array; // never called

}

Nếu **earlyExit** = true, hàm trả về và vùng nhớ cấp phát không bị hủy.

Nếu biến **array** là kiểu vector, vùng nhớ cấp phát tự động bị hủy ngay lập tức khi đi ra ngoài phạm vi hàm (bất chấp hàm có thoát ra sớm hay không)

1. Kích thước mảng

Std::vector có lưu trữ kích thước nội bộ, bạn dùng hàm thành viên **size()** để lấy:

#include <iostream>

#include <vector>

using namespace std;

int main(){

vector<int> array { 9, 7, 5, 3, 1 };

cout << "The size is: " << array.size() << '\n';

return 0;

}

Kết quả:

The size is: 5

1. Co giãn kích thước

Co giãn mảng động truyền thống khá phức tạp với các phần tử là đối tượng. Đây là một nguyên nhân mà bạn nên dùng std::vector.

Bạn dùng hàm thành viên **resize(<kích\_thước>)**

#include <vector>

#include <iostream>

using namespace std;

int main(){

vector<int> array { 0, 1, 2 };

array.resize(5); // set size to 5

cout << "The size is: " << array.size() << '\n';

for (auto const &element: array)

cout << element << ' ';

return 0;

};

Kết quả

The size is: 5

0 1 2 0 0

**Lưu ý: Khi co giãn, giá trị phần tử được bảo lưu.**

**Phần tử mới sẽ được khởi tạo giá trị mặc định (vd: kiểu số là zero)**

1. Nén dữ liệu kiểu bools

Std::vector có một mẹo để nén vùng nhớ mảng các giá trị luận lý.

Đoạn code sau minh họa việc khai báo mảng std::vector kiểu bool chứa 8 chân trị trong 1-bytes nhớ

#include <vector>

#include <iostream>

using namespace std;

int main(){

vector<bool> array { true, false, false, true, true };

cout << "The size is: " << array.size() << '\n';

for (auto const &element: array)

cout << element << ' ';

return 0;

};

Kết quả

The size is: 5

1 0 0 1 1

The way the boolean version of std::vector works behind the scenes is pretty complicated. When you use the subscript Thao tác trên std::vector

1. Thao tác trên vector

Đây là một số thao tác phổ biến :

* **Push\_back(value)** : hàm thành viên chèn một phần tử có giá trị **value** vào cuối vector, tự mở rộng kích thước nếu cần.
* **Size() :** lấy số phần tử của vector

Xem ví dụ sau:

#include <iostream>

#include <vector>

using namespace std;

int main(){

// tao vector kiểu int

vector<int> vec;

int i;

// xuất kích thước bạn đầu của vector

cout << "vector size = " << vec.size() << endl;

// đẩy 5 phần tử vào cuối vector

for(i = 0; i < 5; i++){

vec.push\_back(i);

}

// xuất kích thước sau khi mở rộng của vector

cout << "extended vector size = " << vec.size() << endl;

// xuất giá trị 5 phần tử đầu của vector

for(i = 0; i < 5; i++){

cout << "value of vec [" << i << "] = " << vec[i] << endl;

}

// dùng iterator để truy xuất giá trị các phần tử

vector<int>::iterator v = vec.begin();

while( v != vec.end()) {

cout << "value of v = " << \*v << endl;

v++;

}

return 0;

}

vector size = 0

extended vector size = 5

value of vec [0] = 0

value of vec [1] = 1

value of vec [2] = 2

value of vec [3] = 3

value of vec [4] = 4

value of v = 0

value of v = 1

value of v = 2

value of v = 3

value of v = 4

1. Khai báo mảng hai chiều

Bạn có thể tạo std::vector của std::vectors. Bạn dùng một chút mẹo vặt trong cú pháp :

vector<vector<int>> vector2Chieu;

Nếu bạn muốn khai báo trước kích thước vector , làm như sau:

int dong = 5;

int cot = 5;

vector<vector<int>> vector2Chieu(dong,vector<int>(cot));

vector<vector<int>> vector2Chieu(5,vector<int>(5));

Trong C++11, bạn có thể khởi tạo qua danh sách khởi tạo chuẩn:

vector<vector<int>> vector2Chieu { { 1, 1, 1 }, { 2, 2, 2 } };

1. Tổng kết

Phần này giới thiệu một cách cơ bản về lớp khuôn mẫu std::vector. Trong bài tiếp theo, chúng ta tìm hiểu các tính năng khác của vector, kích thước vùng nhớ và kích thước mảng cũng như cách quản lý vùng nhớ cấp phát.

Với các ưu điểm

* Tự quản lý vùng nhớ
* Bảo lưu nội bộ kích thước mảng
* Khả năng co giãn an toàn

Chúng ta khuyến khích dùng std::vector thay cho mảng động

##### std::array C++11

Mảng tĩnh và động có sẵn trong C++ có điểm dở là: khi tham chiếu mảng bằng con trỏ, thông tin về kích thước của mảng là không biết được, và mảng tĩnh hay có vấn đề khi điều chỉnh kích thước vùng nhớ cấp phát.

Để giải quyết, thư viện chuẩn C++11 đã phát triển một bộ quản lý mảng, std::array và std::vector.

Được biết tới bắt đầu từ C++11, std::array cho phép tạo mảng tĩnh. Với std::array, bạn không dùng con trỏ để tham chiếu khi truyền vào hàm.

std::array được khai báo trong thư viện **<array>**

1. Khởi tạo mảng

Cú pháp khai báo:

std::array < kiểu\_dữ\_liệu, số\_phần\_tử > tên\_biến;

std::array < kiểu\_dữ\_liệu, số\_phần\_tử > tên\_biến = {danh sách khởi tạo};

std::array < kiểu\_dữ\_liệu, số\_phần\_tử > tên\_biến {danh sách khởi tạo chuẩn } ; C++11

Ví dụ khai báo std::array

#include <array>

std::array<int, 3> myarray; // khai báo mảng số nguyên có độ dài là 3

Giống như mảng tĩnh, kích thước std::array phải định nghĩa trước. Std::array cũng có danh sách khởi tạo hay khởi tạo chuẩn C++11.

std::array<int, 5> myarray = { 9, 7, 5, 3, 1 }; // danh sách khởi tạo

std::array<int, 5> myarray2 { 9, 7, 5, 3, 1 }; // khởi tạo chuẩn C++11

Khác với mảng tĩnh, std::array không thể tạo mảng mà chưa khai báo kích thước khi dùng danh sách khởi tạo

std::array<int, > myarray = { 9, 7, 5, 3, 1 }; // error, chưa khai báo kích thước

Ngoài ra bạn có thể dùng toán tử gán cho danh sách khởi tạo

myarray = { 0, 1, 2, 3, 4 }; // okay

myarray = { 9, 8, 7 }; // okay, phần tử 3 và 4 sẽ bằng zero

myarray = { 0, 1, 2, 3, 4, 5 }; // error, danh sách khởi tạo dài hơn kích thước mảng

1. Tự động hủy mảng

Chúng ta không cần gọi phương thức hủy nào cả, std::array sẽ tự hủy khi ra ngoài phạm vi.

1. Truy xuất phần tử với at()

Bạn vẫn dùng toán tử [] để truy xuất phần tử : :

std::cout << myarray[1];

myarray[2] = 6;

Nhưng toán tử [] sẽ không kiểm tra ranh giới phần tử mảng. Bạn nên dùng phương thức .**at(<index>)** thay thế vì nó có kiểm tra ranh giới mảng:

std::array<int, 5> myarray { 9, 7, 5, 3, 1 };

myarray.at(1) = 6; // gán phần tử thứ 1 = 6

myarray.at(9) = 10; // phần tử thứ 9 không có, error

Ở đoạn mã trên, trình biên dịch kiểm tra phần tử thứ nhất có tồn tại không, nếu có nó trả về tham chiếu tới phần tử thứ nhất và gán 6 vào.

Tuy nhiên vì phần tử thứ 9 không có, nó sẽ ném error, ngắt chương trình luôn ( thực ra là nó ném ra một ngoại lệ xử lí lỗi std::out\_of\_range mà chúng ta sẽ nói trong phần [**Exception Handling**](#_C++_-_Error))

1. Số phần tử mảng

Bạn dùng hàm thành viên **size()** để lấy số phần tử mảng.

std::array<double, 5> myarray { 9.0, 7.2, 5.4, 3.6, 1.8 };

std::cout << "size: " << myarray.size();

Kết quả:

size: 5

1. Truyền tham số

Bạn truyền tham số std::array vào hàm như sau:

#include <iostream>

void printSize(const std::array<double, 5> &myarray){

std::cout << "size: " << myarray.size();

}

int main(){

std::array<double, 5> myarray { 9.0, 7.2, 5.4, 3.6, 1.8 };

printSize(myarray);

return 0;

}

Kết quả:

size: 5



**Lưu ý: thành viên size() trả về số phần tử mảng chứ không phải kích thước vùng nhớ của mảng.**

**Bạn luôn truyền tham chiếu hằng std::array vào hàm, để trình biên dịch không tạo bản sao của mảng.**

1. Duyệt mảng

Bạn dùng vòng lặp for-each duyệt std::array

std::array<int, 5> myarray { 9, 7, 5, 3, 1 };

for (auto &element : myarray) //Mẹo là dùng biến auto để trình biên dịch tự nhận diện kiểu dữ liệu

std::cout << element << ' ';

Bạn tham khảo từ khóa auto C++11 [**tại đây**](#_Từ_khóa_auto)

Bạn chú ý vòng lặp for dùng biến hằng tham chiếu const &**element**, để không tạo ra bản sao của phần tử **myarray** và tránh ánh hưởng tới hiệu suất chương trình.

Bạn duyệt mảng dùng for với chiều dài mảng myarray.**size**() cũng được.

1. Sắp xếp

Bạn dùng phương thức std::sort() trong thư viện <algorithm> cùng với iterator để sắp xếp std::array :

#include <iostream>

#include <array>

#include <algorithm> // for std::sort

int main(){

std::array<int, 5> myarray { 7, 3, 1, 9, 5 };

std::sort(myarray.begin(), myarray.end()); // sắp xếp mảng tăng dần

std::sort(myarray.rbegin(),myarray.rend()); // sắp xếp mảng giảm dần

for (const auto &element : myarray)

std::cout << element << ' ';

return 0;

}

Kết quả

1 3 5 7 9

Hàm sort() có sử dụng **iterators,** chúng ta tìm hiểu trong phần khác.

1. Tổng kết

std::array là sự thay thế hoàn hảo cho mảng tĩnh:

* Nó hiệu quả, hỗ trợ nhiều thao tác thuận lợi , nhanh chóng mà không tốn thêm nhiều bộ nhớ.
* Điểm yếu duy nhất là cú pháp khai báo hơi rường rà, phải khai báo số phần tử (hàm dựng đối tượng không tính sẵn cho bạn khi khởi tạo) và không thể co giãn kích thước như std::vector.

##### Std::map

A map is used to store a collection of ordered entries that consists of keys and their values.

Keys must be unique, but several keys can map to the same values. Thus values need not

be unique. The keys in the map are maintained in logically sorted order.

The map behaves like a set instantiated with a pair, whose comparison function

refers only to the key.5 Thus it supports begin, end, size, and empty, but the under

lying iterator is a key-value pair. In other words, for an iterator itr, \*itr is of type

pair<KeyType,ValueType>. The map also supports insert, find, and erase. For insert, one

must provide a pair<KeyType,ValueType> object. Although find requires only a key, the

iterator it returns references a pair. Using only these operations is often not worthwhile

because the syntactic baggage can be expensive.

Fortunately, the map has an important extra operation that yields simple syntax. The

array-indexing operator is overloaded for maps as follows:

ValueType & operator[] ( const KeyType & key );

The semantics of operator[] are as follows. If key is present in the map, a reference to the

corresponding value is returned. If key is not present in the map, it is inserted with a default

value into the map and then a reference to the inserted default value is returned. The default

value is obtained by applying a zero-parameter constructor or is zero for the primitive

types. These semantics do not allow an accessor version of operator[], so operator[] cannot

be used on a map that is constant. For instance, if a map is passed by constant reference,

inside the routine, operator[] is unusable.

The code snippet in Figure 4.68 illustrates two techniques to access items in a map.

First, observe that at line 3, the left-hand side invokes operator[], thus inserting "Pat" and a

double of value 0 into the map, and returning a reference to that double. Then the assignment

changes that double inside the map to 75000. Line 4 outputs 75000. Unfortunately, line 5

inserts "Jan" and a salary of 0.0 into the map and then prints it. This may or may not

be the proper thing to do, depending on the application. If it is important to distinguish

between items that are in the map and those not in the map, or if it is important not to

insert into the map (because it is immutable), then an alternate approach shown at lines

7 to 12 can be used. There we see a call to find. If the key is not found, the iterator is

the endmarker and can be tested. If the key is found, we can access the second item in the

pair referenced by the iterator, which is the value associated with the key. We could also

assign to itr->second if, instead of a const\_iterator, itr is an iterator.