



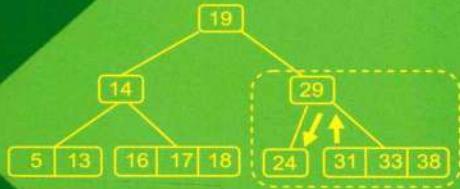
TRẦN THÔNG QUẾ

CK.0000078880

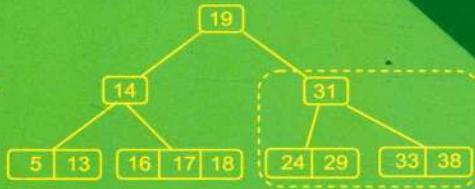
CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN

(PHÂN TÍCH VÀ CÀI ĐẶT TRÊN C/C⁺⁺)

(Tập 2)



Trước khi dịch chuyển



Sau khi dịch chuyển

GUYÊN
LIỆU



NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG

TRẦN THÔNG QUẾ

**CẤU TRÚC
DỮ LIỆU VÀ THUẬT TOÁN**

(PHÂN TÍCH VÀ CÀI ĐẶT TRÊN C/C++)

(Tập 2)

NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG

LỜI NHÀ XUẤT BẢN

Trong thời đại bùng nổ công nghệ thông tin như hiện nay, dữ liệu đang được sinh ra ngày càng nhiều và dưới nhiều hình thức khác nhau. Việc xử lý để ứng dụng kho dữ liệu quý giá đó như thế nào để đạt được hiệu quả cao nhất luôn là vấn đề được rất nhiều những chuyên gia về công nghệ thông tin đặc biệt quan tâm. Chính vì lý do đó mà *Cấu trúc dữ liệu và thuật toán* vẫn luôn là một chủ đề mà ngày càng được đầu tư nghiên cứu và phát triển; không bị lỗi thời trong bất cứ xu hướng nào của lĩnh vực Công nghệ thông tin, việc phát triển đề thuật toán ngày một hoàn thiện hơn gần như chưa bao giờ dừng lại và thậm chí là không có giới hạn.

Trước nhu cầu thiết thực cần tìm hiểu của bạn đọc, tác giả Trần Thông Quέ - giảng viên cao cấp đã có trên 35 năm kinh nghiệm giảng dạy ở lĩnh vực Công nghệ thông tin và luôn tâm huyết với chủ đề này đã phối hợp với Nhà xuất bản Thông tin và Truyền thông xuất bản bộ sách "*Cấu trúc dữ liệu và thuật toán - Phân tích và cài đặt trên C/C++*", gồm 2 tập.

Tập 1 gồm 5 chương:

Chương 1: Các khái niệm cơ bản về thuật toán và cấu trúc dữ liệu

Chương 2: Các thuật toán tìm kiếm trong và sắp xếp trong

Chương 3: Một số chiến lược thiết kế thuật toán

Chương 4: Các kiểu dữ liệu trừu tượng và biến nhớ động

Chương 5: Cấu trúc cây

Tập 2 gồm 4 chương:

Chương 1: Sắp xếp ngoài

Chương 2: Phép băm và bảng băm

Chương 3: Cây đỏ đen

Chương 4: Cây 2-3-4 và B-cây

Với kinh nghiệm nhiều năm giảng dạy, tác giả đã lựa chọn cách tiếp cận đơn giản hóa các vấn đề phức tạp, từ cơ bản đến nâng cao. Nội dung cuốn sách được trình bày một cách chi tiết, dễ hiểu, sau mỗi nội dung đều có ví dụ minh họa, sau mỗi thuật toán đều được đánh giá độ phức tạp và cài đặt demo. Tác giả bổ sung thêm một số thuật toán mà những cuốn sách đã xuất bản trước đây chưa có như: Thuật toán sắp xếp rung lắc, thuật toán sắp xếp với độ dài bước giảm dần, cây 2-3-4, B-cây... Chính vì vậy, cuốn sách này hướng tới đối tượng độc giả đông đảo, từ những bạn đọc mới làm quen tới những bạn đọc đã có kinh nghiệm nhiều năm trong lĩnh vực này. Đây thực sự là tài liệu bổ ích dành cho sinh viên, giảng viên, lập trình viên cũng như các chuyên gia về công nghệ thông tin có nhu cầu tìm hiểu và nghiên cứu.

Mọi ý kiến góp ý của bạn đọc xin gửi về địa chỉ: Nhà xuất bản Thông tin và Truyền thông; tầng 6, tòa nhà 115 Trần Duy Hưng, Hà Nội. Điện thoại: 024.35772143; Fax: 024.35579858.

Xin trân trọng giới thiệu cùng bạn đọc./.

NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG

LỜI NÓI ĐẦU

Cấu trúc dữ liệu và Thuật toán (Data Structure and Algorithms) là môn học bắt buộc không những với mỗi sinh viên ngành Công nghệ Thông tin mà còn là môn học bắt buộc và môn thi quốc gia đầu vào bắt buộc với các nghiên cứu viên của ngành học đó. Nó là một trong các môn học khó của ngành CNTT, đặc biệt càng khó đối với hầu hết các sinh viên khi phải cài đặt một thuật toán hay một bài toán nào đó thuộc môn học này.

Dựa trên thực tế đó, bằng kinh nghiệm tích lũy được trong nhiều năm liên tục giảng dạy lý thuyết và dạy thực hành môn học “Data Structure and Algorithms” chúng tôi biên soạn cuốn sách này nhằm đáp ứng một cách tối ưu cho mỗi sinh viên, các nghiên cứu viên ngành CNTT và cả các giảng viên phụ trách bộ môn này với tâm niệm giảng dạy còn chưa nhiều.

Bất cứ sự thành công nào của một dự án Tin học đều là kết quả của việc kết hợp khéo léo giữa Cấu trúc dữ liệu và Thuật toán. Khẳng định này được chứng tỏ trong một công thức rất ngắn gọn mang tính triết lý đương đại nghề nghiệp:

BIG DATA + BIG COMMUNITY = BIG RESULT

Việc cài đặt các thuật toán cơ bản hoặc nổi tiếng hoặc khó được thực hiện trên các ngôn ngữ chuyên nghiệp đương đại là C/C⁺⁺ nhằm giải đáp câu hỏi tồn tại trong đầu những người học là: Thực thi các thuật toán ấy trên máy tính điện tử như thế nào và sẽ cho kết quả ra sao?

Chúng tôi tin tưởng rằng tài liệu này sẽ rất bổ ích và hữu hiệu cho tất cả những ai có nhu cầu tìm hiểu về cấu trúc dữ liệu và giải thuật - một môn học tuy phức tạp song không kém phần lý thú, hấp dẫn.

Mặc dù đã có nhiều năm giảng dạy môn học này ở không ít trường đại học công lập và dân lập, song do những hạn chế chủ quan của bản thân nên

cuốn sách sẽ khó tránh khỏi những khiêm khuyết, rất mong các độc giả xa gần cho các ý kiến “phản biện” để các lần tái bản sau sẽ ngày càng hoàn chỉnh hơn.

Xin chân thành cảm ơn các độc giả!

Tác giả
Trần Thông Quέ

Chương 1

SẮP XẾP NGOÀI

Sắp xếp ngoài là sắp xếp trật tự (tăng hoặc giảm) cho các dữ liệu tồn tại trên bề mặt các bộ nhớ ngoài (băng từ, đĩa cứng, đĩa quang...) được lưu giữ dưới dạng tệp tin (File). Các bộ nhớ ngoài luôn có sức chứa (dung lượng) lớn hơn rất nhiều dung lượng các bộ nhớ trong. Việc truy xuất dữ liệu trên các bộ nhớ ngoài lại rất chậm so với việc truy xuất dữ liệu ở các bộ nhớ trong. Do hai đặc điểm này mà việc sắp xếp ngoài sẽ khó hơn sắp xếp trong và cần các phương pháp với đặc thù riêng.

Dưới đây ta sẽ lần lượt tìm hiểu một số phương pháp ấy.

1.1. PHƯƠNG PHÁP XẾP TRỘN CÁC RUN

1.1.1. Một số khái niệm cơ bản về run

- **Định nghĩa run:** Một dãy liên tiếp các phần tử được sắp xếp có trật tự (giảm hoặc tăng) gọi là *run*. Ví dụ 1.1: 34 25 18 15 11 9 8 là một run có 7 phần tử.

- *Dãy chỉ có một phần tử độc nhất* cũng được xem là một run.

- *Độ dài của run* là số phần tử thuộc run. Ở ví dụ 1.1 độ dài run là 7.

- *Phương pháp xếp trộn hai run:* Từ 2 run đã cho tạo ra một run mới gọi là phép trộn run (Merge Sort Run).

1.1.2. Thuật toán xếp trộn hai run

Thuật toán này gồm các bước sau:

- f_v là tệp vào chứa các dữ liệu (chưa có trật tự) cần sắp xếp có trật tự (giảm hoặc tăng dần).

- f_r là tệp ra mà dữ liệu trên nó đã có trật tự (tệp ra).

- Các tệp f_1, f_2 là các tệp trung gian (các tệp tạm) để lưu tạm thời các dữ liệu phân phối từ tệp vào lên chúng.

Tất cả các tệp nói trên đều có thể là các tệp văn bản (Text File) hoặc các tệp nhị phân (Binarry File).

a. Các bước thực thi thuật toán

Tệp vào cho trước là $f_v = 33 \ 16 \ 55 \ 44 \ 68 \ 40 \ 19 \ 37 \ 26$

Tất nhiên trước khởi sự thì các tập f_1, f_2, f_r đều rỗng!

Bước 1: + Thực hiện phân phối chỉ một lần các run có độ dài $L=1$ từ f_v lên các run tạm f_1 và f_2 :

$$\begin{aligned}f_1 &= \underline{33} \ 55 \ 68 \ 19 \ 26 \\f_v &= \underline{33} \ \underline{16} \ 55 \ 44 \ 68 \ 40 \ 19 \ 37 \ 26 \\f_2 &= \underline{16} \ \underline{44} \ 40 \ 37\end{aligned}$$

+ Trộn f_1 với f_2 để được f_r :

$$f_r = 16 \ 33 \ 44 \ 55 \ 40 \ 68 \ 19 \ 37 \ 26 \quad \text{Đặt } f_r \text{ này là } f_v \text{ (mới)}$$

Bước 2: Phân phối các run với độ dài $L=2 (=2^1)$ lên các file f_1 và f_2 :

$$\begin{aligned}f_1 &= \underline{16} \ \underline{33} \ 40 \ 68 \ 26 \\f_v &= \underline{16} \ \underline{33} \ \underline{44} \ \underline{55} \ 40 \ 68 \ 19 \ 37 \ 26 \\f_2 &= \underline{44} \ \underline{55} \ 19 \ 37\end{aligned}$$

Trộn hai file (run) trên để được f_r và đặt nó là f_v mới cho bước kế tiếp:

$$f_v = 16 \ 33 \ 44 \ 55 \ 19 \ 37 \ 40 \ 68 \ 26$$

Bước 3: Phân phối các run với độ dài $L=4 (2^2)$ lên các file f_1 và f_2 :

$$\begin{aligned}f_1 &= \underline{16} \ \underline{33} \ \underline{44} \ \underline{55} \\f_v &= \underline{16} \ \underline{33} \ \underline{44} \ \underline{55} \ \underline{19} \ \underline{37} \ \underline{40} \ \underline{68} \ 26 \\f_2 &= \underline{19} \ \underline{37} \ \underline{40} \ \underline{68}\end{aligned}$$

Trộn hai file (run) trên để được f_v mới cho bước kế tiếp:

$$f_v = 16 \ 19 \ 33 \ 37 \ 40 \ 44 \ 55 \ 68 \ 26$$

Bước 4: Phân phối các run với độ dài $L=8 (2^3)$ lên các file f_1 và f_2 :

$$\begin{aligned}f_1 &= \underline{16} \ \underline{19} \ \underline{33} \ \underline{37} \ \underline{40} \ \underline{44} \ \underline{55} \ \underline{68} \\f_v &= \underline{16} \ \underline{19} \ \underline{33} \ \underline{37} \ \underline{40} \ \underline{44} \ \underline{55} \ \underline{68} \ \underline{26} \\f_2 &= \underline{26}\end{aligned}$$

Trộn hai file (run) trên để được f_v mới (dừng ở đây thì f_v này chính là tệp ra f_r đã sắp).

$$f_v = f_r = 16 \ 19 \ 26 \ 33 \ 37 \ 40 \ 44 \ 55 \ 68$$

Tiếp tục thao tác trên cho tới khi chiều dài L của các run cần phân phối lên các tệp tạm f_1 và f_2 vượt quá độ dài n của dây đã cho thì dừng. Và khi ấy run f_v cuối cùng chính là run ra f_r đã được sắp thứ tự.

(Trong ví dụ trên, thuật toán trộn dừng ở cuối bước 3 (vì đến bước 4 thì độ dài các run cần trộn là $L \geq n=9$ của dây đã cho).

b. *Dánh giá độ phức tạp của thuật toán run*

- Nếu độ dài của dây vào là N thì cần $\log_2 N$ bước (vì mỗi lượt xử lý run có độ dài L tăng gấp đôi).

- Tại mỗi bước:

+ Để phân phối phải thực hiện copy N lần

+ Để trộn cần so sánh $N/2$ lần.

- Vật tồng thể cần:

+ Copy $2N * \log_2 N$ lần

+ So sánh $(N/2) * \log_2 N$ lần.

c. *Cài đặt*

```
#include "iostream.h"
#include "iomanip.h"
#include "conio.h"
#include "stdio.h"
#define MAX 100
int b[MAX], c[MAX], nb, nc; // nb: độ dài run b; nc: độ dài run c
void Phanphoi(int a[], int N, int &nb, int &nc, int k)
    //Phân phối các phần tử của dây ban đầu cho các dây con b và c
    (int i, ja, jb, jc; //i, ja: chỉ số dây đã cho;
     jb, jc: chỉ số các dây con.
```

```

ja=jb=jc=0;
while (ja<N)      //Chừng nào chưa duyệt hết dãy
    ban đầu thì lặp:
    {for (i=0; (ja<N) && (i<k); i++, ja++, jb++)
//Duyệt theo chiều dài dãy ấy, lặp:
    b[jb]=a[ja]; //Phân phôi p_tù a[ja] lên
                    dãy b tại jb
    for (i=0; (ja<N) && (i<k); i++, ja++, jc++)
//Duyệt theo chiều dài dãy ban đầu, rồi lặp lại các thao
tác sau:
    c[jc]=a[ja];      //Phân phôi p_tù a[ja] lên
                        dãy c tại jc
    }
    nb=jb;           // Cập nhật jb
    nc=jc;           // Cập nhật jc
}
int min(int a, int b)//Hàm tìm min trong 2 số cho trước.
{
if(a>b) return b;
else return a;
}
void Tron(int a[], int nb, int nc, int k) //Thực thi
                                             thuật toán trộn
{int j, jb, jc, ib, ic, kb, kc; //j: chỉ số dãy ban đầu;
jb, jc, ib, ic: chỉ số các dãy con b, c.
j=jb=jc=0; ib=ic=0;
while((nb>0)&&(nc>0))      //Chừng nào chưa xét hết
    các run b, c thì lặp lại:
    {kb=min(k,nb); kc=min(k,nc); //Tim số min trong
        các run b, c
    if(b[jb+ib]<=c[jc+ic]) //Nếu phần tử c[jc+ic]
                            ≥ b[jb+ib] thì
        {a[j++]=b[jb+ib]; ib++; //tăng j lên 1 rồi lưu
        (trộn) b[jb+ib] vào a[j], sau đó tăng ib lên 1
        if(ib==kb) //Nếu ib=kb thì lặp lại các việc sau:
        {for(;ic<kc;ic++) //Duyệt theo độ dài run c, lặp:

```

```

        a[j++]=c[jc+ic];      //tăng j lên 1, rồi lưu
        (trộn) c[jc+ic] vào a[] tại j
        jb+=kb; //Cập nhật jb
        jc+=kc; //Cập nhật jc
        ib=ic=0; //Tái khởi trị ib, ic
        nb-=kb; //Cập nhật nb
        {Sau trộn độ dài các
         run b, c biến đổi một
         lượng tương ứng là kb, kc.
        nc-=kc; //Cập nhật nc
        }
    }

else //Ngoài lại (b[jb+ib] > c[jc+ic]) tăng j lên 1 rồi
    {a[j++]=c[jc+ic]; ic++; //trộn c[jc+ic] vào a
     tại j; tăng ic lên 1
     if(ic==kc)           //Nếu ic==kc (thì)
     {for(;ib<kb;ib++) //Duyệt theo độ dài run b,
      lặp:
        a[j++]=b[jb+ib]; //tăng j lên 1, rồi trộn
                           b[jb+ib] vào a tại j
        jb+=kb;          //Cập nhật jb
        jc+=kc;          //Cập nhật jc
        ib = ic=0;        //Tái khởi trị ib, ic
        nb-=kb;          //Cập nhật nb
        nc-=kc;          //Cập nhật nc
        }
     }
    }
}

void XepTron(int a[], int N) //Hàm này gọi hàm
    Phanphoi và gọi hàm Tron để sắp xếp ngoài
{ int k; nc=0; nb=0;
  for (k = 1; k < N; k *= 2)
  {Phanphoi(a, N, nb, nc, k);
   Tron(a, nb, nc, k);
}

```

```

    }
}

void Nhap(int a[], int n) //Hàm nhập dữ liệu
{ int i;
    for(i=0; i<n; i++)
        {cout<<"Nhập a["<<i<<"]:";
        cin>>a[i];
        }
}

void Xuat(int a[], int n) //Hàm xuất dữ liệu
{ int i;
    for(i=0; i<n; i++)
        cout<<setw(4)<<a[i];
}

void main() /* Hàm chính, cho biết dãy vừa nhập và
{int a[MAX],n; clrscr(); xem kết quả trộn run */
    cout<<"Nhập số phần tử n của dãy:";
    cin>>n;
    Nhap(a, n);
    cout<<"Đã nhập xong!"<<endl;
    Xuat(a, n); cout<<endl<<endl;
    XepTron(a, n);
    cout<<"Kết quả xep trộn dãy vừa nhập:";
    Xuat(a, n);
    getch();
}

```

1.1.3. Thuật toán trộn tự nhiên

Trong thuật toán trộn run nói trên ta phân phối lần lượt các run (lên các file tạm) với độ dài ban đầu là $L = 1$ rồi tăng dần gấp đôi độ dài ấy. Cách làm này rõ ràng làm chậm việc chạy chương trình. Nhược điểm ấy sẽ càng thể hiện rõ nếu độ dài dãy ban đầu lớn hoặc rất lớn.

Để việc trộn run hiệu quả hơn người ta dùng thuật toán trộn tự nhiên. Trong thuật toán này ta trộn các run có độ dài lớn nhất với nhau cho tới khi đạt được run kết quả: dãy đã được sắp.

a. Các bước của thuật toán

f_v : file dữ liệu vào cần sắp xếp

f_r : file kết quả đã sắp

- Pha 1: Phân phối dữ liệu lên các file tạm f_i :

+ Bước 1: Phân phối (sao chép) các run có độ dài lớn nhất lên file tạm f_i .

Ta dùng biến EoR (End of Run = hết run) báo hết run với trị
 $EoR = 1$

+ Bước 2: Sao chép run kế tiếp có độ dài lớn nhất lên tệp tạm f_{i+1}

+ Bước 3: Thao tác phân phối kết thúc khi đã xét hết tệp vào f_v .

- Pha 2: Trộn một run ở f_i và một run ở f_{i+1} vào f_r

Thao tác trộn kết thúc khi đã có đủ n phần tử của file vào f_v đã được sao
lên f_r . Và tất nhiên lúc này file f_r đã được sắp xếp.

b. Cài đặt

- Yêu cầu: So với cài đặt trên, lần này sẽ đặt thêm những đòi hỏi nhỏ và có
thể là “khó hơn” với một số không ít sinh viên. Các đòi hỏi ấy là:

- Về thao tác nhập dữ liệu vào chương trình, yêu cầu cung cấp 2 cách cho
người dùng:

+ Nhập bằng tay

+ Nhập tự động

- Việc hiển thị (Display): Yêu cầu hiển thị các kết quả phân phối trung gian
(không dễ với nhiều sinh viên) và tất nhiên hiển thị kết quả chạy chương trình.

- Khuyến cáo:

+ Hãy tự lực cài đặt thuật toán này theo các yêu cầu trên trong 1/2 ngày
(khoảng 12 tiếng liền) mà không làm được, lúc ấy hãy gõ và test code dưới đây.

+ Trước tiên hãy dùng cách nhập dữ liệu bằng tay với bộ dữ liệu vào sau:

17 1 23 41 68 55 12 77 46

+ Lần kiểm thử sau bạn hãy chọn cách 2 (tự động nhập dữ liệu ngẫu nhiên, các lần kiểm thử tiếp là tùy theo ý của bạn!)

```
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#define true 1
#define false 0
#define sz sizeof(float)
int EoF(FILE *f) //Hàm báo hết tệp
{
    float x;
    if(fread(&x, sz, 1, f)<=0) return(true);
    fseek(f, -1.0*sz, 1);
    return(false);
}
int EoR(FILE *f) //Hàm báo hết run
{
    float x, y;
    if(ftell(f)==0) return(false);
    if(fread(&y, sz, 1, f)<=0) return(true);
    fseek(f, -2.0*sz, 1);
    fread(&x, sz, 1, f);
    if(x<=y) return(false);
    else return(true);
}
int FileNumMem(char*TenTep) //Hàm trả về số phần tử
    của tệp
{
    int k;float x;
    FILE*f=fopen(TenTep, "rb");
    fseek(f, 0, 2);
    k=f.tell(f)/sz
    fclose(f);
    return(k);
};
void CreateFile(char*TenTep) //Tạo file có n phần tử
```

```
{int i,m; float x;
FILE*f;
f=fopen(TenTep, "wb");
rewind(f);
char ch;
do
{ clrscr();
printf("\n Nhập dữ liệu vào tệp :");
printf("\n1 Nhập trực tiếp bằng tay:");
printf("\n2 Tao dữ liệu ngẫu nhiên, tự động cho
chuỗi trinh.");
printf("\n\n Hãy chọn 1 hoặc 2:");
ch=getche();
}
while(ch!='1'&& ch!='2');
printf("\n Cho biết số phần tử cần đưa vào tệp:");
scanf("%d", &m);
if(ch=='1')
{printf("\n Hãy nhập %d số:", m);
for(i=0; i<m; i++)
{scanf("%f", &x);
fwrite(&x, sz, 1, f);
}
}
else
{randomize();
for(i=0; i<m; i++)
{x=float(random(10*m));
fwrite(&x, sz, 1, f);
}
}
fclose(f);
};
```

```

void ViewFile(char*TenTep)    //Hiển thị nội dung file
                            lên màn hình
{
    float x;
    FILE*f;
    f=fopen(TenTep,"rb");
    rewind(f);
    printf("Cac phan tu tren tep %s:\n\n",TenTep);
    while(fread(&x, sz, 1, f)>0)
        printf("%5.0f", x);
    fclose(f);
};

Void Phanphoi(char*tepA,char*tepB,char*tepC)
//Phân phối lần lượt từng run trên file a cho file b và
cho file c đến khi hết file a.
{
    FILE*a,*b,*c;float x;
    a=fopen(tepA,"rb");
    b=fopen(tepB,"wb");
    c=fopen(tepC,"wb");
    rewind(a); rewind(b); rewind(c);
    while(!EoF(a))
//Vòng lặp phân phối một run trên a cho b
    {
        fread(&x, sz, 1, a);
        fwrite(&x, sz, 1, b);
        while(!EoR(a))
            {fread(&x, sz, 1, a);
             fwrite(&x, sz, 1, b);
            }
        if(EoF(a)) break;
//Nếu hết file a thì kết thúc công việc
        fread(&x, sz, 1, a);
        fwrite(&x, sz, 1, c);
        while(!EoR(a))
//Vòng lặp phân phối một run trên a cho c

```

```

        { fread(&x, sz, 1, a);
        fwrite(&x, sz, 1, c);
    }
}

fclose(a);fclose(b);fclose(c);
};

//Dưới đây là hàm trộn một run trên b với một run trên c
//thành run trên a cho đến khi kết thúc file b và kết thúc
//file c

Void MergeFile(char*tepB,char*tepC,char*tepA)
//Hàm trộn các run
{
    FILE *a,*b,*c;
    float x,y; int res_x, res_y;      //res_x, res_y :
Theo thứ tự ấy, báo hiệu còn hay hết run trên tệp b, tệp c
    b=fopen(tepB,"rb");
    c=fopen(tepC,"rb");
    a=fopen(tepA,"wb");
    rewind(a); rewind(b); rewind(c);
    while(!EoF(b)&&!EoF(c))
    {
        fread(&x, sz, 1, b);
        res_x=true; //Còn phần tử trong run của tệp b
        fread(&y, sz, 1, c);
        res_y=true; //Còn phần tử trong run của tệp c
        while(res_x && res_y)
        {
            if(x<y)
                { fwrite(&x, sz, 1, a);
                  //nếu sau khi ghi x vào a mà hết
                  //run trên file b thì biến nhớ res_x=false
                if(EoR(b)) res_x=false; else fread(&x, sz, 1, b);
            }
            else //x>=y
                { fwrite(&y, sz, 1, a);
                  //nếu sau khi ghi y lên a mà hết run trên
                  //file c thì biến nhớ res_y=false
                }
        }
    }
}

```

```

        if(EoR(c))  res_y=false;  else fread(&y,sz,1,c);
    }
}

//Hết run trên b hoặc trên c phân phối phần còn lại
của run trên b hoặc c lên a
if(res_x)  fwrite(&x, sz, 1, a);
if(res_y)  fwrite(&y, sz, 1, a);
}

//copy phần còn lại trên b lên a
while(!EoF(b))
{
    fread(&x, sz, 1, b);
    fwrite(&x, sz, 1, a);
}

//copy phần còn lại trên c lên a
while(!EoF(c))
{
    fread(&x, sz, 1, c);
    fwrite(&x, sz, 1, a);
}

fclose(a);fclose(b);fclose(c);
};

int SortedFile(char*TenTep)           //Hàm trả về true
nếu tệp đã sắp, ngược lại trả về false
{
FILE*f;float x,y;
f=fopen(TenTep,"rb");
rewind(f);
fread(&x, sz, 1, f);
while(fread(&y, sz, 1, f)>0)
{
    if(x>y)  { fclose(f);return(false); }
    x=y;
}
fclose(f);
return(true);
}

```

```

void HamGoiHam(char*TenTep)      //Hàm này gọi các hàm
                                  Phanphoi, ViewFile và MergeFile
{
//để thực thi xếp trộn tự nhiên
    int i;char*tepB="Tep_b.dat",*tepC="Tep_c.dat";
    while(!SortedFile(TenTep))
    {
        Phanphoi(TenTep,tepB,tepC);
        ViewFile(tepB);
        printf("\n");
        ViewFile(tepC);
        printf("\n");
        MergeFile(tepB,tepC,TenTep);
    }
    remove("Tep_b.dat");
    remove("Tep_c.dat");
}
void main()
{
    clrscr();
    char *TenTep="Tep_a.DAT";
    CreateFile(TenTep);
    ViewFile(TenTep);
    printf("\n\n CAC BUOC PHAN PHOI TEP VA KET QUA XEP
TRON:\n");
    HamGoiHam(TenTep);
    ViewFile(TenTep);
    getch();
}

```

1.2. THUẬT TOÁN SẮP XẾP TRỘN ĐA ĐƯỜNG CÂN BẰNG

Hai thuật toán đã xét trên tốn nhiều thời gian cho việc phân phối các run ở file vào lên các file tạm, vì vậy hiệu quả thuật toán chưa cao (thời gian thực thi lớn khi dãy vào dài hoặc cực dài (hàng vạn phần tử trở lên). Thuật toán sắp xếp trộn đa đường cân bằng (Balanced MultiWay Merging-BMM) phần nào khắc phục được nhược điểm của hai thuật toán đã xét. Sở dĩ như vậy vì thuật toán BMM

chi thực hiện 1 giai đoạn trộn thay vì dùng 2 giai đoạn như trên, nên tiết kiệm được 1/2 thời gian sao chép (song tất nhiên BMM lại phải dùng số lượng file tạm tăng gấp đôi → không gian nhớ ngoài tăng gấp đôi).

1.2.1. Mô tả ngắn gọn các bước của BMM bằng ngôn ngữ tự nhiên

+ Ký hiệu $F_s = \{f_1, f_2, f_3, \dots, f_n\}$ là tập NGUỒN chứa các tệp nguồn ($s \rightarrow$ source: nguồn)

+ Ký hiệu $F_t = \{g_1, g_2, g_3, \dots, g_n\}$ là tập ĐÍCH chứa các tệp đích ($t \rightarrow$ target: đích)

- Bước 1: Phân phối luân phiên mỗi lần 1 run thuộc file f_0 cần sắp xếp lên các file f_i ($i=1, n$) thuộc tập nguồn F_s cho đến khi f_0 không còn phần tử nào.

- Bước 2: Trộn từng bộ run của các file $\in F_s$ để có run mới (với độ dài lớn hơn các run thành phần) và ghi lên các file $f_j \in F_t$ ($j = 1, n$)

- Bước 3: Nếu số run ở các file của F_t còn nhiều hơn 1 thì:

- Hoán đổi vai trò của tập nguồn F_s với tập đích F_t .

- Quay lại bước 2.

Ngược lại kết thúc thuật toán.

Ví dụ 1.2: Cho $f_0 = \{81 \ 94 \ 11 \ 96 \ 12 \ 35 \ 17 \ 99 \ 28 \ 58 \ 41 \ 75 \ 15\}$

Với ví dụ này ta dùng 3 file nguồn f_i ($i = 1, 2, 3$) và 3 file đích f_j ($j = 1, 2, 3$) và quy trình làm theo thuật toán trên như sau:

+ Phân phối lần 1:

$$f_1 = \{11 \ 81 \ 94\}$$

$$f_2 = \{41 \ 58 \ 75\}$$

$$f_3 = \{12 \ 35 \ 96\}$$

+ Trộn lần 1:

$$g_1 = \{11 \ 81 \ 94 \ 41 \ 58 \ 75\}$$

$$g_2 = \{12 \ 35 \ 96\}$$

$$g_3 = \{17 \ 28 \ 99\}$$

+ Phân phối lần 2:

$$f_1 = \{11 \ 81 \ 94 \ 41 \ 58 \ 75\}$$

$f_2 = \{12 \ 35 \ 96 \ 15\}$

$f_3 = \{17 \ 28 \ 99\}$

+ Trộn lần 2:

$g_1 = \{11 \ 12 \ 17 \ 28 \ 35 \ 81 \ 94 \ 96 \ 99\}$

$g_2 = \{15 \ 41 \ 58 \ 75\}$

$g_3 = \{\} \ //Rỗng$

+ Phân phối lần 3: Lần này đơn giản, nên chúng ta không liệt kê mà thực hiện trộn lần 3 luôn:

+ Trộn lần 3:

$g_1 = \{11 \ 12 \ 15 \ 17 \ 28 \ 35 \ 41 \ 58 \ 75 \ 81 \ 94 \ 96 \ 99\}$

$g_2 = \{\}$

$g_3 = \{\}$

Kết thúc.

1.2.2. Mô tả chi tiết thuật toán BMM bằng mã giả

a. Các ký hiệu

+ $F_s = \{f_1, f_2, f_3, \dots, f_n\}$: Tập NGUỒN chứa các file nguồn (N)

+ $F_t = \{g_1, g_2, g_3, \dots, g_n\}$: Tập ĐÍCH chứa các file đích (D)

+ F_{Mer} : Tập các file nguồn tham gia trộn để tạo một run cho file đích (S)

+ F_{Cur} : Tập các file nguồn còn đang xử lý (Q)

+ F_{Spl} : Tập các file đích đã được phân phối một run trong lượt hiện hành.

Khi tất cả các file đích đã được phân phối 1 run thì hết một lượt hiện hành (D)

b. Các bước của thuật toán BMM

Bước 1:

//Bắt đầu

Phân phối luân phiên các run của dãy ban đầu F (file chưa có trật tự, cần sắp xếp) cho các file thuộc F_s .

i=1

while (!feof(F))

```

Phân phối 1 run của F lên f_i
    i=(i mod n)+1;
FMer=Fs;      //Tắt cả các file nguồn đều tham gia vào
                quá trình trộn
FSp1={};      //Các file đích chưa được phân phối run
run_num=0;    //Khởi trị cho biến đếm số run trên các
                tệp đích

```

Bước 2:

2.1. Lấy một file đích trong tập hiệu (F_t-F_{Sp}) – tập các file đích trong lượt hiện hành chưa được phân phối run, ta gọi nó là file đích hiện hành F_{hh} .

F_{Curr}= F_{Mer}; //F_{curr}: tập các file hiện hành; F_{hh}: file hiện hành

2.2. $x_{\min} = \text{Min}\{x_a \mid a \in F_{\text{Curr}}\};$ // x_a là một phần tử thuộc file a của tập F_{Curr} .

2.3. Copy x_{amin} lên F_{hh}

2.4. if (file nguồn a hết)

1

```

Fcurr = Fcurr - {a};           //Số còn lại các file
                                đang cần xử lý
FMer = FMer - {a};           //Số còn lại các file tham
                                gia trộn
if(FMer-{})                  /*Nếu hết các file tham gia
                                trộn → kết thúc việc trộn
                                từ tập nguồn tới tập đích */
run_num++;                   //(thì) bắt đầu đếm
Esol=Esol + {Ebb};

```

Sang bước 3; //Hoán đổi vai trò các file nguồn và đích Es ↔ Et

1

else

(if (F_{CYFR} ≠ {})

Quay lại bước 2.2

```
else (hiểu ngầm là Fcurr = {}) //Các bộ run đã trộn xong), thực hiện đếm
```

```

        run_num++;
        Fspl = Fspl + {Fhh};
        if (Fspl==Ft) ; //Xong một lượt trộn
        Fspl = 0; //Cập nhật giá trị cho Fspl
        Quay lại bước 2.1;
    }
}
}

else (ngầm hiểu file nguồn a chưa hết run)
{
    if (trên a hết run)
    {
        Fcurr = Fcurr - {a}; //Số tệp nguồn còn lại sau
                               khi xử lý xong a
        if (Fcurr != {});
        Quay lại bước 2.2;
    }
    else //Fcurr khác rỗng
    {
        run_num++;
        Fspl = Fspl +{Fhh};
        if(Fspl == Fs);
        Fspl = {}; //Xong một lượt phân phối run
        Quay lại bước 2.1;
    }
}
else //Run hiện tại trên a chưa hết, tiếp tục tìm xmin
    Quay lại bước 2.2;
}
}

```

Bước 3:

```

if(run_num==1)      STOP;
else
    if(run+num<n) Fmer = Fspl; //Không đủ run để phân
                                  phối cho n file
    else Fmer = Ft;
    Fspl = {};
    run_num=0;
}

```

```
Fs ↔ Ft; //Hoán đổi Tập các file nguồn và file đích
Quay lại bước 2; //Kết thúc
```

1.2.3. Cài đặt

Đọc giả hãy dựa vào Pseudocode chi tiết trên đây cài đặt cho thuật toán BMM (xem như một bài tự luyện)

1.3. THUẬT TOÁN TRỘN ĐA PHA

1.3.1. Mở đầu

Trong chiến lược sắp xếp trộn đa đường cân bằng, nếu trộn k đường thì cần $2k$ tệp để thực thi việc này. Như vậy thuật toán đó “ngốn” khá nhiều không gian bộ nhớ ngoài. Sở dĩ tồn tại nhược điểm này vì trong BMM, một nửa số file luôn giữ vai trò trộn (các file nguồn) và nửa còn lại số các file làm nhiệm vụ phân phối (các file đích). Để khắc phục nhược điểm ấy của BMM người ta dùng thuật toán trộn đa pha (Polyphase Merge Sort-PMS). Trong thuật toán trộn đa pha chỉ cần $k+1$ tệp bởi vì trong chiến lược này người ta thay đổi vai trò của các file trong cùng một lượt xử lý.

1.3.2. Mô tả thuật toán PMS bằng ngôn ngữ tự nhiên

Để trực giác, ta xét ví dụ sắp xếp dãy f_0 với 3 file f_1, f_2, f_3 . Các bước thực thi việc này như sau:

Bước 1: Phân phối luân phiên các run của f_0 lên các file f_1, f_2 .

Bước 2: Trộn các run của f_1 và f_3 với nhau rồi cắt lên f_3 . Thuật toán kết thúc nếu f_3 chỉ có 1 run.

Bước 3: Chép nửa số run của f_3 vào f_1 .

Bước 4: Trộn các run của f_1 và f_3 vào f_2 . Thuật toán kết thúc nếu f_2 chỉ có 1 run.

Bước 5: Chép nửa số run của f_2 vào f_1 . Quay lại bước 2.

Nhược điểm của phương pháp này là tốn thời gian chép nửa số run từ file này sang file kia. Ta khắc phục nhược điểm này bằng cách tiến sau: đầu tiên bắt đầu với f_n run của file f_1 và f_{n-1} của file f_2 , với f_n và f_{n-1} là hai số liên tiếp của dãy FIBONACI mà chúng ta đã tìm hiểu kỹ ở tập 1 của cuốn sách này. Các file f_n và f_{n-1} liên hệ với nhau bởi công thức F dưới đây:

$$f_n = f_{n-1} + f_{n-2} \quad (F)$$

1.3.3. Ví dụ cách cài tiến của PMS nhờ dùng hệ thức (F)

Giả sử ta cần xếp dãy cho trước (đang lộn xộn) có 34 run. Quy trình xếp dãy này dựa vào (F) mô tả bởi bảng dưới đây:

	Phân chia ban đầu	Bước 1	Bước 2	Bước 3	Bước 4	Bước 5	Bước 6	Bước 7
f1	0	13	5	0	3	1	0	1
f2	21	8	0	5	2	0	1	0
f3	13	0	8	3	0	2	1	0

Khi chỉ còn 1 run
thuật toán kết thúc!

Chương 2

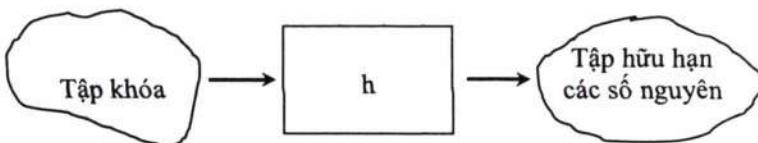
PHÉP BĂM VÀ BẢNG BĂM

Những thao tác xử lý (sắp xếp, tìm kiếm, xóa, chèn thêm...) dữ liệu trên các cấu trúc (danh sách, stack, queue, heap, cây nhị phân tìm kiếm...) đã trình bày ở tập 1 bộ sách này đa số dựa vào phép so sánh giá trị các phần tử trong cấu trúc. Vì vậy thời gian thực thi các thao tác ấy chậm và phụ thuộc vào kích cỡ của cấu trúc. Để giảm nhỏ tối mức có thể được thời gian thực thi các thao tác trên, người ta nghiên cứu phép băm và bảng băm. Bảng băm là cấu trúc dữ liệu trùu tượng đáp ứng yêu cầu nêu trên.

2.1. CÁC KHÁI NIỆM CƠ BẢN

2.1.1. Phép băm hay hàm băm

Phép băm hay hàm băm (Hashing or Hashing Function, viết tắt: h) là ánh xạ 1-1 từ tập các khóa (key) vào tập hữu hạn các số nguyên (hình 2.1). Các số nguyên ấy gọi là các **địa chỉ** của bảng băm. Dùng các địa chỉ này để truy xuất dữ liệu.



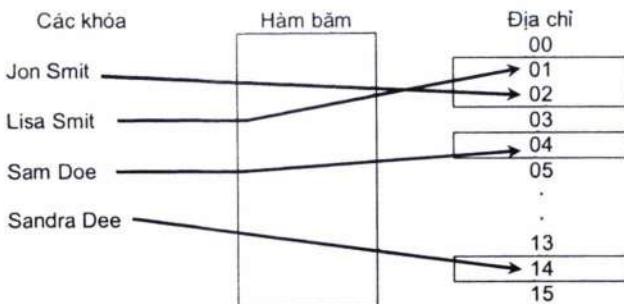
Hình 2.1. Phép băm

Khóa có thể là các ký tự hoặc số.

2.1.2. Bảng băm

Bảng băm (Hash Table) là một cấu trúc dữ liệu trùu tượng (Abstract Data Type) chứa kết quả của phép băm (gồm tập khóa và tập các địa chỉ tương ứng nói ở mục trên).

Ví dụ 2.1: Hình 2.2 cho ta một hình ảnh về bảng băm



Hình 2.2. Bảng băm

2.1.3. Các chỉ tiêu về một hàm băm tốt

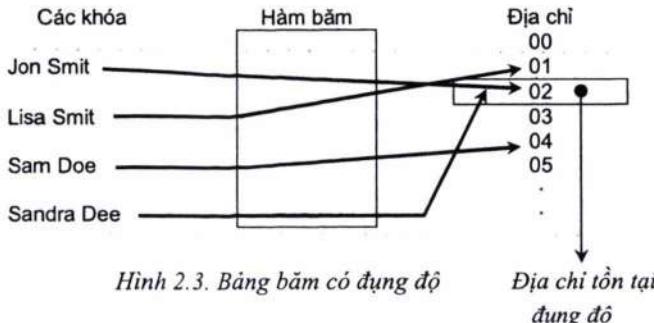
Một hàm băm tốt phải thỏa mãn các yêu cầu sau:

- Tính toán nhanh.
- Các khóa được phân bổ đều trên bảng băm.
- Hạn chế tối đa xảy ra đụng độ.

2.1.4. Độ dung sai trong bảng băm là gì?

Độ dung sai là hiện tượng các khóa khác nhau được ánh xạ vào cùng một địa chỉ.

Ví dụ 2.2: Hình 2.3 là một bảng băm có độ dung sai



Hình 2.3. Bảng băm có độ dung sai

Địa chỉ tồn tại
đụng độ

2.1.5. Một vài loại hàm băm diễn hình

2.1.5.1. Hàm băm hoàn chỉnh

Một hàm băm gọi là hàm băm hoàn chỉnh (Perfect Hashing Function) nếu nó ánh xạ mỗi khóa với một (và chỉ một) địa chỉ.

Với hàm băm này, ta có thể định vị trực tiếp (directly locate) một cách chính xác một khóa bất kỳ trong bảng băm (tạo bởi hàm ấy) mà không cần bắt cứ tìm kiếm hỗ trợ nào!

Bảng băm sinh bởi hàm băm nói trên gọi là bảng băm hoàn chỉnh. Bảng băm ở hình 2.2 là một trong các bảng băm hoàn chỉnh.

Ví dụ về một hàm băm như vậy được đề xuất bởi D. Bernstein sử dụng phép XOR:

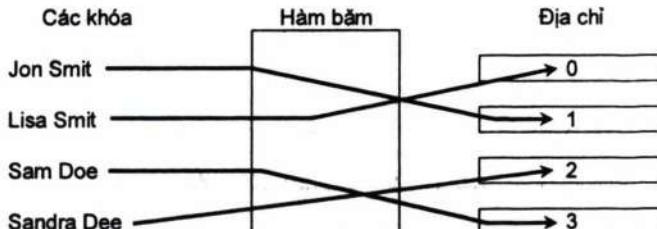
```
hash(i) = hash(i-1)*33 ^ str[i] //trong đó str[] là mảng ký tự
```

Điều kỳ diệu và “bí ẩn” ở đây là chính con số 33, với con số này, hàm băm của Bernstein làm việc tốt hơn bất kỳ với hàng số nào khác. Cho đến nay vẫn chưa có một giải thích thỏa đáng nào và dù thuyết phục nào về con số kỳ lạ 33 trong công thức trên.

2.1.5.2. Hàm băm hoàn chỉnh tối thiểu

Một hàm băm hoàn chỉnh dùng cho M khóa được gọi là hàm băm hoàn chỉnh tối thiểu (Minimal Perfect Hashing Function) nếu miền địa chỉ của nó chứa M số nguyên liên tiếp gồm từ 0 đến M-1.

Hàm băm hoàn chỉnh tối thiểu tạo sinh bảng băm hoàn chỉnh tối thiểu. Một bảng băm như thế minh họa ở hình 2.4.



Hình 2.4. Bảng băm Minimal Perfect

2.1.5.3. Hàm băm phân bố đều dữ liệu

Trong những trường hợp đầu vào (khóa) là một xâu ký tự có độ dài hạn chế (ví dụ như số di động, tên các đại lý bán ô tô, địa chỉ các bệnh viện...) xuất hiện độc lập với xác suất đều thì ta cần một hàm băm ánh xạ “làm tròn” (roughly) các khóa ấy với mỗi giá trị băm. Một ví dụ dễ hiểu về trường hợp này là tập khóa k gồm N số nguyên từ 0 đến N-1 và đầu ra (giá trị băm (địa chỉ)) là tập h các số nguyên từ 0 đến M-1 (với $N > M$). Khi đó hàm băm có thể là $h(k)=k \bmod M$ hoặc có thể dùng một vài công thức khác (M : kích thước của bảng băm, nói rõ hơn là số địa chỉ trong bảng băm).

Ở công thức trên, việc chọn M là cả một vấn đề không nhỏ. Qua thử nghiệm thực tế, người ta thấy chọn M là số nguyên tố gần với 2^i là giá trị tốt (theo nghĩa ít xung đột và phân bố đồng đều dữ liệu). Chẳng hạn nếu bảng băm gồm 4000 khóa thì chọn $M=4093$.

Hàm băm như vậy gọi là hàm băm phân bố đều dữ liệu (Hashing Uniformly Distributed Data).

2.1.5.4. Hàm băm phổ quát

Ta gọi U là tập các số nguyên không âm. Khi đó tập H các hàm băm phổ quát (Universal Hashing Function) được xác định như sau:

$$H = \{h(k) : U \rightarrow \{0, 1, 2, \dots, M-1\} \quad M: \text{kích thước bảng băm}$$

H được gọi là các hàm băm phổ quát nếu tồn tại $x \neq y$ mà $\{h() \in H : h(x) = h(y)\} = |H|/M$ trong đó $|H|$ là số phần tử trong H (hay còn gọi là **bản số** của tập H).

Để đảm bảo $\forall x \neq y$ mà $h(x) = h(y) = |H|/M$ thì xác suất để chọn hàm $h \in U$ là:

$$P((h(x) = h(y))) = \frac{|H|/M}{|H|} = \frac{1}{M} \quad (*)$$

Ý nghĩa của hệ (*) là ở chỗ: Muốn làm không hơn $1/M$ thay đổi xung đột giữa 2 khóa phân biệt x và y thì các hàm $h(x)$ và $h(y)$ phải được chọn một cách ngẫu nhiên và độc lập từ tập $\{0, 1, 2, \dots, M-1\}$.

2.1.5.5. Thiết kế các hàm băm phổ quát như thế nào?

Việc thiết kế hàm băm phổ quát được thực hiện như sau:

- Chọn số nguyên tố p đủ lớn sao cho khóa $k \in$ miền $[0 \dots p-1]$ và xác định các tập Z_p và Z_p^* như sau:

$$Z_p = \{0, 1, \dots, p-1\} \text{ và } Z_p^* = \{1, \dots, p-1\}$$

- Đến đây ta định nghĩa hàm băm như sau:

$$h_{a,b}(k) = ((ak+b) \% p) \% M \text{ với } \forall a \in Z_p^* \text{ và } b \in Z_p$$

Bộ (family) tất cả các hàm băm như thế được xác định bởi hệ thức:

$$\mathcal{H}_{p,M} = \{h_{a,b} : a \in Z_p^* \text{ và } b \in Z_p\}$$

a, b được chọn một cách ngẫu nhiên tại lúc bắt đầu thực thi công việc.

2.1.5.6. Ví dụ về thiết kế hàm băm phổ quát

Cho $p = 17$, $M = 6$, $k = 8$; chọn $a = 3$, $b = 4$ ta có:

$$\begin{aligned} h_{a,b}(k) &= ((a \cdot k + b) \% p) \% M = h_{3,4} = ((3 \cdot 8 + 4) \% 17) \% 6 \\ &= (28 \% 17) \% 6 = 11 \% 6 = 5 \end{aligned}$$

2.1.5.7. Code C mô tả một số thao tác cơ bản trên bảng băm nói chung

a. Định nghĩa bảng băm nhờ danh sách liên kết đơn

```
struct node
{
    int key;
    node* next;
};

node *H[M]; //Bảng băm là một mảng các pointer
            //trỏ đến các danh sách
```

b. Khởi tạo bảng băm (Initiate)

Gán trị ban đầu là rỗng cho bảng băm:

```
void init()
{
    for(int i=0; i<M; i++)
        init(H[i]);
}
```

c. Thiết lập hàm băm cho bảng băm

Thường phải chọn hàm sao cho hạn chế tốt nhất sự dụng độ (sẽ bàn kỹ ở các mục sau). Giả sử ta chọn hàm băm là phép chia:

$$h(k) = k \% M \quad \left\{ \begin{array}{l} \text{Trong đó } k \text{ là khóa, còn } M \text{ là kích thước bảng băm} \end{array} \right.$$

Khi đó hàm băm là:

```
int h(int k)
{
    return (k%M);
}
```

d. Kiểm tra rỗng (Empty)

Kiểm tra bảng băm có rỗng hay không:

```
int empty()
{
    for(int i=0;i<M;i++) if(!empty(H[i]))
        return (false);
    return (true); //Nếu bảng băm rỗng thì hàm empty trả
                   //về true
}
```

e. Thêm (Chèn) khóa mới vào bảng băm (Insert) nhờ đệ quy

```
void insert(int x) //Chèn thêm khóa mới vào hashtable
                   //dùng đệ quy
{
    int k=h(x);
    insert(H[k],x);
}
```

f. Tìm một khóa bất kỳ trong bảng băm (Search) nhờ đệ quy

```
node* search(int x)
{
    int k=h(x); node* p;
    p=search(H[k],x);
    return (p);
}
```

g. Hủy (Delete) một khóa trong bảng băm nhờ đệ quy

```
void delete(int x)
{
    int k=h(x);
    delete(H[k],x);
}
```

h. Duyệt bảng băm (Traverse)

```
void traverse()
{int i;
printf("\n\nDia_chi   Cac_khoa:");
for(i=0;i<M;i++) {printf("\t\n %3d: ",i);
traverse(H[i]);}
}
```

i. Xóa toàn bộ bảng băm (giải phóng vùng nhớ đã cấp phát cho các nút trong bảng băm)

```
void clear()
{for(int i=0;i<M;i++) clear(H[i]);}
```

Chú ý: Đây không phải là cách duy nhất để mô tả các thao tác trên bảng băm với code C, bạn có thể dùng C++ với cấu trúc dữ liệu là Class để thực hiện việc đó.

2.2. CÁC PHƯƠNG PHÁP GIẢI QUYẾT DỤNG ĐỘ TRONG BẢNG BĂM

Hiện nay có một vài phương pháp giải quyết dung độ trong bảng băm là:

- Kết nối phân nhóm (Separate Chaining):

- + Kết nối trực tiếp (Direct Chaining)

- + Kết nối hợp nhất (Coalesced Chaining)

- Địa chỉ mở (Open Addressing):

- + Dò tuyến tính (Line Probing)

- + Dò bậc hai (Quadratic Probing)

- + Băm kép (Double hashing)

Dưới đây chúng ta lần lượt xét các phương pháp này.

2.2.1. Tạo bảng băm bằng phương pháp kết nối trực tiếp

a. Tổ chức dữ liệu

Với phương pháp này, bảng băm được biểu diễn bởi danh sách liên kết đơn như sơ đồ hình 2.5.

Cho kích thước bảng băm là M. Khi ấy bảng băm được biểu diễn bởi danh sách liên kết đơn (DSLKĐ) trong đó các phần tử thuộc bảng băm được băm thành M danh sách liên kết. Các DSLKD ấy được đánh số từ 0 đến M-1. Các phần tử xung đột ở địa chỉ i được kết nối trực tiếp với nhau trong danh sách liên kết đơn thứ i. Ví dụ các phần tử xung đột ở địa chỉ 5 thì xếp vào DSLKD có chỉ số i=5.

Khi chèn thêm một phần tử có khóa k vào bảng băm thì hàm băm $h(k)$ sẽ xác định địa chỉ i trong phạm vi từ 0 đến M-1 ứng với danh sách liên kết i mà phần tử đó sẽ được thêm vào.

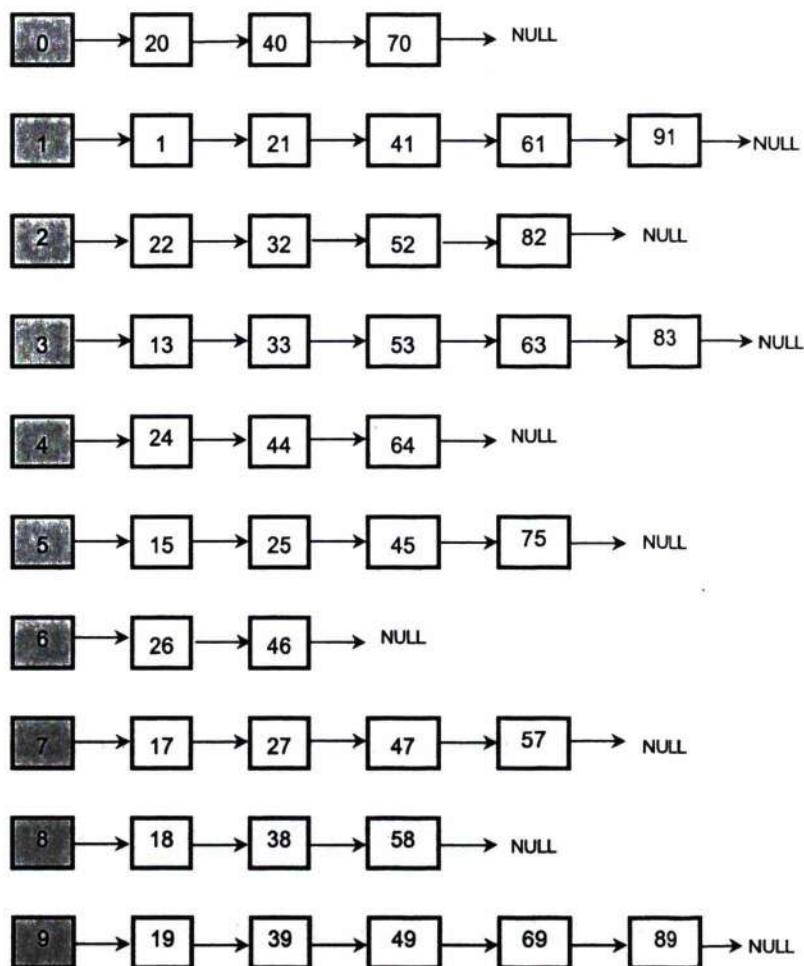
Khi tìm một phần tử có khóa k thì hàm băm cũng xác định địa chỉ i trong khoảng từ 0 đến M-1 ứng với danh sách i có thể chứa phần tử này. Rõ ràng việc tìm kiếm một phần tử trong bảng băm thực chất là tìm kiếm một phần tử trong DSLKD.

Bảng băm ở hình 2.5 có kích thước $M = 10$. Hàm băm của bảng băm này là $h(k) = k \% 10$.

Dưới đây là cài đặt để thực thi bảng băm với phương pháp kết nối trực tiếp đã trình bày ở trên.

b. Cài đặt bảng băm dùng phương pháp kết nối trực tiếp

```
#include<stdlib.h>
#include<conio.h>
#include<stdio.h>
#include<ctype.h>
#define M 10
//-----
```



Hình 2.5. Bảng băm dùng danh sách liên kết đơn
với phương pháp kết nối trực tiếp

```
struct node
{
    int key;
    node* next;
};
```

```

node *H[M]; //Bảng băm là một mảng gồm các pointer
             trỏ đến các danh sách
//-----
int h(int x); //Hàm băm ánh xạ khóa x lên vị trí
               tương ứng trong bảng băm
void clear(node* &phead); //Xóa toàn bộ danh sách
                           (giải phóng vùng nhớ cấp phát cho danh sách)
void init(node* &phead); //Khởi tạo danh sách do con
                           trỏ đầu danh sách phead trỏ đến
int empty(node* &phead); //Kiểm tra danh sách có rỗng
                           không
node* search(node* &phead,int x); //Tim khóa x trên
                                   danh sách do phead trỏ đến
void insert(node* &phead,int x); //Chèn (thêm) khóa x
                                   vào danh sách do phead trỏ đến
void delete(node* &phead,int x); //Xóa khóa x trên
                                   danh sách do phead trỏ đến
void traverse(node* &phead); //Duyệt danh sách do
                           phead trỏ đến
void clear(); //Xóa bảng băm
void init(); //Khởi tạo bảng băm
int empty(); //Kiểm tra xem bảng băm có
              rỗng không
node* search(int x); //Tim khóa x trong bảng băm
void insert(int x); //Chèn khóa x vào bảng băm
void delete(int x); //Xóa khóa x trong bảng băm
void traverse(); //Duyệt bảng băm
//-----
void clear(node* &phead)
{
    node *p,*p1;
    p=phead;
    while(p!=NULL)
    {
        p1=p;
        p=p->next;
    }
}

```

```

        delete p1;
    }
    phead=NULL;
}
//-----
void init(node* &phead)
{
    clear(phead);
    phead=NULL;
}
//-----
int empty(node* &phead)
{
    return (phead==NULL);
}
//-----
node* search(node* &phead, int x)
{
    node* p=phead;
    while(p!=NULL) //Chừng nào chưa rà soát hết
                    danh sách
    {
        if(p->key==x) return(p); //mà nếu giá trị
        //khóa do p trả về bằng x thì trả về
        //địa chỉ của p tức là tìm thấy x
        p=p->next; //Xét nút kế tiếp trong danh sách
    }
    return(NULL); //Đã rà soát hết danh sách
}
//-----
void insert(node* &phead, int x)
{
    node *pp,*p,*p1;
    p=search(phead,x); //Trước tiên phải tìm xem x đã
    //có trong danh sách chưa, nếu có thì không chèn x được
}

```

```

    if(p!=NULL) (printf("\nKhoa %4d da co, khong chen
duoc!",x);return;}
    pp = new node;           //Cấp phát vùng nhớ cho con trỏ
                            //liên kết pp
    pp->key=x;   pp->next=NULL;      //Chèn giá trị x vào
                            //trường khóa key do pp trỏ đến
    if(phead==NULL) {phead=pp; return;} //Nếu danh sách
                            //rỗng thì lưu địa chỉ của pp vào phead
    p1=NULL;                 //Khởi tạo trống cho p1
    p=phead;                //Cho p trỏ tới đầu danh sách
    while(p!=NULL)          //Chừng nào chưa xét hết danh
                            //sách (thì)
    {
        p1=p;               //Cho p1 trỏ tới đầu danh sách
        p=p->next;          //Chuyển tới nút kế tiếp
    }
    p1->next = pp; //Kết thúc vòng lặp thì p1 trỏ
                    //tới nút cuối cùng:
};

//-----
void delete(node* &phead,int x)
{
    node *p,*p1,*q;
    if(phead==NULL) return; //Nếu danh sách rỗng thì
                            //không xóa được
    if(phead->key==x) //Nếu trường khóa key do phead
                        //trỏ tới có giá trị =x thì
        {p=phead; phead=phead->next; delete p; return;}
//xóa x bằng cách xóa vùng nhớ đã cấp cho phead
    p1=NULL;           //Cập nhật địa chỉ (rỗng) cho p1
    p=phead;            //Cập nhật địa chỉ cho p
    while(p!=NULL)      //Chừng nào danh sách chưa rỗng
                        //(thì).
    {
        if(p->key==x) //Nếu trường key do p trỏ tới có
                        //khóa = x
            {q=p; p1->next=p->next; delete q; return;}
//(thì) cập nhật địa chỉ q, xóa q, xét nút kế tiếp
}
}

```

```

        p1=p;      //Cập nhật địa chỉ cho p1
        p=p->next; //Cho p trỏ tới phần tử tiếp theo
    }
    printf("\nKhoa %4d khong co, khong xoa duoc!");
}
//-----
void traverse(node* &phead)
{
    node* p=phead;
    while(p!=NULL) //Chừng nào danh sách không rỗng thì
                    //chừng đó
        { printf("%5d",p->key); //ta còn in lần lượt các
          //phần tử của nó!
            p=p->next;
        }
}
//-----
int h(int x)
{
    return(x%M); //Phép toán xác định hàm băm
}
//-----
void init()
{
    for(int i=0;i<M;i++) init(H[i]); //Khởi tạo bảng băm
}
//-----
void clear()           //Xóa toàn bộ bảng băm
{
    for(int i=0;i<M;i++) clear(H[i]);
}
//-----
int empty()
{

```

```

        for(int i=0;i<M;i++) if(!empty(H[i])) //Nếu bảng
                                băm không rỗng thì hàm kiểm tra empty()
            return(false);                      //trả về false
            return(true);           //ngược lại nó trả về true
        }
//-----
node* search(int x)
{ int k=h(x); node* p;
    p=search(H[k],x);      /*Cho con trỏ p trỏ tới hàm
                           search() để tìm x.
    return(p);           Nếu x có trong bảng băm thì
                           trả về địa chỉ của con trỏ p*/
}
//-----
void insert(int x)
{ int k=h(x);
    insert(H[k],x);     //Dùng đệ quy để thêm khóa x vào
                         bảng băm
}
//-----
void delete(int x)
{ int k=h(x);
    delete(H[k],x); //Dùng đệ quy để xóa khóa x trong
                     bảng băm
}
//-----
void traverse()
{ int i;
    printf("\n\nDia_chi * * * Cac_khoa:");
    for(i=0;i<M;i++) {printf("\t\n %3d: ",i);
        traverse(H[i]); }      //In toàn bộ bảng băm
    }
//-----

```

```

void CachNhap() //Hai cách thêm một phần tử vào
                bảng băm
{
    clrscr(); int m,i,x; init();
    printf("\nNhap du lieu vao bang bam:");
    printf("\n1. Nhap truc tiep");
    printf("\n2. Tao ngau nhien");
    printf("\n\n Hay chon 1 hoac 2: ");
    fflush(stdin);
    char ch=getch();
    printf("\nCho biet so khoa can nhap vao bang bam: ");
    fflush(stdin);
    scanf("%d",&m);
    if(ch=='1')
        {
            printf("\nHay nhap %d phan tu: ",m);
            for(i=0;i<m;i++)
                {
                    scanf("%d",&x);
                    insert(x);
                }
        }
    else
        {
            randomize();
            for(i=0;i<m;i++)
                {
                    x=random(10*m);
                    insert(x);
                }
        }
};

//-----
void main()
{
    clrscr(); int x,k; node* p;
    while(true)           //Đoạn chương trình tạo Menu
    {
        clrscr();
        printf("\n 1. Chen cac khoa vao bang bam");
}

```

Chèn các
khóa x vào
bảng băm
bằng tay

Tự động chèn
các khóa mới x
vào bảng băm

```

printf("\n 2. Tim kiem khoa tren bang bam");
printf("\n 3. Xoa khoa tren bang bam ");
printf("\n 4. Duyet bang bam ");
printf("\n 0. Go 0 de ket thuc");
printf("\n Go mot trong cac ky tu [1..0] de chon viec:");
");

char chon=getch();
if(chon=='0') break;
switch(chon)
{
    case '1':CachNhap();break;
    case '2':printf("\nKhoa can tim la:");scanf("%d",&x);
    k=h(x);p=search(x);
    if(p!=NULL)
        printf("\nKhoa tim thay o vi tri %d la: %d",k,p->key);
    else printf("\nKhong tim thay");break;
    case '3':printf("\nNhap khoa muon xoa:");
    fflush(stdin);scanf("%d",&x);
    delete(x);break;
    case '4':if(empty()) printf("\nBang bam rong");
    else traverse();break;
}
printf("\n\nNhan phim bat ky de tiep tuc");
getch();
}
}

```

c. Phân tích tốc độ xử lý của bảng băm dùng phương pháp kết nối trực tiếp

Trong bảng băm dùng phương pháp kết nối trực tiếp người ta băm n phần tử vào M danh sách liên kết. Muốn tốc độ thực hiện các phép toán trên bảng băm không chậm thi ta cần chọn hàm băm sao cho băm đều n phần tử vào M DSLKD. Khi đó một cách trung bình, mỗi danh sách có n/M phần tử. Giả sử ta

thực hiện phép tìm kiếm tuyến tính trên mỗi danh sách. Trong trường hợp này độ phức tạp của phép tìm kiếm là $O(n/M)$. Nghĩa là nhanh gấp n lần khi tìm kiếm tuyến tính trên danh sách có n phần tử.

Nhưng nếu chọn M càng lớn thì tốc độ tìm kiếm càng nhanh, song lại tốn nhiều bộ nhớ. Do đó cần chọn M sao cho dung hòa được giữa đòi hỏi về tốc độ xử lý và kích thước vùng nhớ bị chiếm dụng.

- Nếu chọn $M = n$ thì tốc độ xử lý tương đương với tốc độ truy xuất trên mảng một chiều, cụ thể là độ phức tạp của việc này là $O(1) = \text{const}$, tuy nhiên hơi tốn bộ nhớ.

- Nếu chọn $M = n/k$ thì tốc độ chậm đi k lần song bộ nhớ tốn ít hơn k lần.

2.2.2. Tạo bảng băm bằng phương pháp kết nối hợp nhất

a. Tổ chức dữ liệu

Trong trường hợp này bảng băm được cài đặt bởi DSKĐ được biểu diễn bởi mảng có M phần tử (là M DSKĐ). Các phần tử xung đột kết nối với nhau trong một DSKĐ.

Mỗi nút trong DSKĐ này có 2 trường (Field):

- Trường key chứa giá trị khóa
- Trường next chứa con trỏ liên kết (để trỏ tới phần tử kế ngay sau)

b. Một số thao tác cơ bản trên bảng băm kết nối hợp nhất (Coalesced Chaining Method - CCM)

- Khởi tạo: Ban đầu mọi trường key và mọi trường next đều gán giá trị -1. Khi đã hình thành bảng băm thì các trường next rỗng cũng được gán -1.

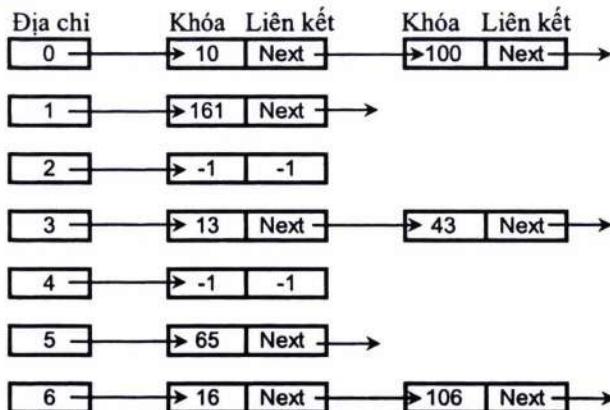
- Thêm một phần tử mới: Trong thao tác này, trước hết hàm băm $h(x)$ sẽ xác định địa chỉ i trong khoảng từ 0 đến $M-1$:

- + Nếu chưa bị xung đột thì thêm phần tử đó vào địa chỉ i này.

- + Nếu bị xung đột thì thêm phần tử ấy vào nút trống (đứng ngay sau khóa xung đột cuối cùng của DSKĐ ứng với địa chỉ i) do con trỏ next trỏ tới.

- + Tìm kiếm: Trước tiên hàm băm $h(x)$ sẽ xác định địa chỉ i trong phạm vi $0 \rightarrow M-1$ và việc tìm kiếm sẽ bắt đầu từ địa chỉ i .

Ví dụ: ta có bảng băm với tập khóa là các số tự nhiên; tập địa chỉ M gồm 7 địa chỉ (0, 1... , 6); hàm băm là $h(x)=k \% 10$. Một bảng băm như thế có cấu trúc như hình 2.6.



Hình 2.6. Bảng băm dùng phương pháp kết nối hợp nhất

c. Cài đặt

```

typedef struct                         //Cấu trúc nút
{
    int key;
    int next;
}node;
node bbam[M];                         //Cấu trúc bảng băm
int chotrong;

+ Khởi tạo bảng băm:
void init()
{
    int i;
    for(i=0; i<M; i++)
        { bbam[i].key=nullkey; //Ban đầu các trường khóa
và các trường next đều gán -1 (nghĩa là rỗng)
        bbam[i].next=-1;
    }
    chotrong=M-1;
}

```

+ Kiểm tra xem bảng băm có rỗng không:

```
int empty()
{ int i;
    for(i=0; i<M; i++)
        if (bbam[i].key!=nullkey)           //Nếu bảng băm
            không rỗng thì hàm empty() trả về giá trị false
        return 1;
    return 0;   //ngược lại hàm ấy trả về true
}
```

+ Tìm một phần tử trong bảng băm:

```
int search(int k)
{ int i;
    i=hf(k);    //Trước tiên hàm băm xác định địa chỉ i
    while (k!=bbam[i].key && i!=-1)    //Chừng nào chưa
        tìm thấy khóa k và địa chỉ i không rỗng
        i=bbam[i].next;                  //thì tiếp tục tìm
    if(k==bbam[i].key)    //Nếu tìm thấy khóa k tại i
        return i;      //thì hàm search() trả về vị trí i đó
    else return (M); //Ngược lại (tại i không có
                      //khóa k) thì trả về rỗng (địa chỉ
    )               //của nút sau nút cuối cùng trong bảng băm)
```

+ Cập nhật vị trí trống trong bảng băm: (để có chỗ chèn phần tử bị xung đột (với các phần tử đã tồn tại) trong DSLKD)

```
int getempty()
{ while (bbam[chotrong].key!=nullkey)           //Chừng nào
    trường khóa key trong bảng băm không rỗng
    chotrong--; //thì ta giảm vị trí chotrong đi một
                 //đơn vị (và)
    return (chotrong); //hàm getempty trả về địa chỉ
                       //của chỗ trống
}
```

+ Chèn thêm khóa mới vào bảng băm: Đây là thao tác phức tạp nhất trong các thao tác đã nêu trên. Vì vậy các bạn cần chú ý các dòng phân tích bên cạnh mã tương ứng!

```

int them(int k)
{ int i,j;
  i=search(k); //Trước tiên phải xác định địa chỉ i
cần chèn khóa mới vào
  if(i!=M)//Nếu tại i đã có giá trị khóa muốn chèn rồi
    { printf("Da co khoa %d nay o bbam, khong them
duoc.",k); // thì không chèn nữa
    return i; // (và) hàm them() trả về địa chỉ i đó!
    }
  i=hf(k); //tiếp theo hàm băm hf(k) xác định vị
trí i khác để chèn
  while (bbam[i].next>=0) //Chừng nào trường next
không rỗng
    i=bbam[i].next; //thì gán địa chỉ của con trỏ
next cho i
  if (bbam[i].key==nullkey) //Nếu trường key
rỗng thì
    j=i; //lưu địa chỉ i vào biến tạm j
  else //ngược lại gọi hàm getempty để tạo vị
trí trống
  {
    j=getempty();
    if (j<0) //Nếu bảng băm đầy thì (cũng) không
chèn được
    { printf("Bang bam day, khong them duoc.");
      return j; //Khi ấy trả về địa chỉ j báo hiệu
bảng băm đầy
    }
    else bbam[i].next=j; //Còn thì (bảng băm không
đầy) ta lưu địa chỉ j vào trường next
    } //của địa chỉ i
  bbam[i].key=k; //(rồi) chèn thêm khóa k vào trường
key của địa chỉ i
  return j;
}
//Xong việc chèn!

```

+ Xem toàn bộ bảng băm:

```
void view()
{ int i;
    printf("DIA CHI KHOA:\n\n");
    for (i=0; i<M; i++)
        printf("\n %d %9d", i, bbam[i].key);
    printf("\n\n");
```

}

+ Xóa toàn bộ bảng băm: Để code ngắn gọn, trong chương trình mẹ (main()) bạn chỉ cần gọi lại hàm init() để xóa sạch bảng băm (chuẩn bị cho việc thử một bảng băm mới)

d. Cài đặt bảng băm dùng phương pháp kết nối hợp nhất

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define true 1
#define false 0
#define nullkey -1
#define M 10

typedef struct
{ int key;
    int next;
}node;

node bbam[M];
int chotrong;

int hf(int key)
{ return (key%10); }

void init()
{ int i;
    for(i=0; i<M; i++)
```

```
    { bbam[i].key=nullkey;
      bbam[i].next=-1;
    }
    chotrong=M-1;
}

int empty()
{int i;
 for(i=0; i<M; i++)
 if (bbam[i].key!=nullkey)
   return 1;
 return 0;
}

int search(int k)
{ int i;
 i=hf(k);
 while (k!=bbam[i].key && i!=-1)
   i=bbam[i].next;
 if(k==bbam[i].key)
   return i;
 else return (M); }

int getempty()
{ while (bbam[chotrong].key!=nullkey)
   chotrong--;
 return (chotrong); }

int them(int k)
{ int i,j;
 i=search(k);
 if(i!=M)
   { printf("Da co khoa %d nay o bbam, khong them
duoc.",k); }
```

```
        return i;
    }
    i=hf(k);
    while (bbam[i].next>=0)
        i=bbam[i].next;
    if (bbam[i].key==nullkey)
        j=i;
    else
    {
        j=getempty();
        if (j<0)
            { printf("Bang bam day, khong them duoc.");
              return j;
            }
        else bbam[i].next=j;
    }
    bbam[i].key=k;
    return j;
}

void view()
{ int i;
  printf("DIA CHI KHOA:\n\n");
  for (i=0; i<M; i++)
    printf("\n %d %9d",i,bbam[i].key);
  printf("\n\n");
}

void main()
{ clrscr();int i,n,key,cnang;char c;
  init();
  while (1)
  { clrscr();
```

```
printf("Cac chuc nang:\n");
printf("1.Them khoa bang tay vao bang bam:\n");
printf("2.Them khoa tu dong, ngau nhien vao bang
bam:\n");
printf("3.Xoa toan bang bam:\n");
printf("4.Duyet toan bang bam:\n");
printf("5.Tim khoa trong bang bam:\n");
printf("0.Thoat\n");
printf("Ban chon chuc nang nao?:");
scanf("%d",&cnang);
if (cnang==0) break;
switch(cnang)
{
    case 1:
        printf("Them khoa bang tay vao bang bam:\n");
        printf("Khoa cua nut moi:");
        scanf("%d",&key);
        them(key); break;
    }
    case 2:
        printf("Them tu dong, ngau nhien khoa vao
bang:\n");
        printf("Can them may khoa?:");
        scanf("%d",&n);
        for (i=0;i<n;i++)
        {
            key=random(300);
            them(key);
        }
        break;
    }
case 3: { printf("Xoa toan bang bam:\n");
    printf("Co chac chan ban muon xoa khong?:");
    c=getch();
```

```

        if(c=='c' || c=='C')
            init();
        break;
    }

    case 4:{ printf("Xem toan bang bam:\n");
        view();
        break;
    }

    case 5: { printf("Tim kiem khoa trong bang:\n");
        printf("Cho gia tri khoa muon tim:");
        scanf("%d",&key);
        if(search(key)==M)
            printf(" Khoa %d khong co trong bang
                bam.\n\n",key);
        else
            printf("Tim thay khoa %d o vi tri:%d\n",
                key,search(key));
        break;
    }
    printf("Go phim bat ky de tiep tuc... \n");
    getch();
}
}

```

e. Xét hiệu quả của phương pháp kết nối hợp nhất

Phương pháp này chỉ tối ưu khi băm đều. Nói rõ hơn: mỗi DSLK chỉ chứa một vài phần tử bị xung đột. Khi đó độ phức tạp (ĐPT) của truy xuất trên bảng băm là $O(1) = \text{const}$. Trường hợp tồi tệ nhất là khi băm không đều. Lúc đó hình thành các DSLK có n phần tử xung đột và hiển nhiên ĐPT của truy xuất là $O(n)$. Nếu n cực lớn (cỡ hàng triệu trở lên) thì tốc độ truy xuất rất chậm!

2.2.3. Tạo bảng băm dùng phương pháp dò tuyến tính

Hàm băm trong phương pháp dò tuyến tính (Linear Probing Method - LPM) hoạt động theo cơ chế truy xuất địa chỉ kế tiếp.

a. Tô chức dữ liệu

Với phương pháp này, bảng băm được cài đặt bởi danh sách kè có M phần tử mà mỗi phần tử là một bản ghi (record) có 1 trường chứa khóa. Ban đầu mọi trường khóa đều được khởi trị là NULL.

b. Một số thao tác cơ bản

- Chèn thêm khóa mới: Trong thao tác này, trước tiên hàm băm $h(x)$ xác định địa chỉ i thuộc phạm vi $[0...M-1]$:

+ Nếu không bị xung đột thì thêm khóa mới này vào vị trí i đó.

+ Nếu bị xung đột thì hàm băm sẽ băm lại lần 1 (ứng với lần băm lại này ta ký hiệu hàm băm là h_1); tiếp theo hàm h_1 xét địa chỉ kế tiếp, nếu bị xung đột thì hàm này băm lại lần 2 (ta ký hiệu là h_2), rồi hàm h_2 xét địa chỉ kế tiếp... Quá trình tiếp tục như vậy cho đến khi tìm được địa chỉ trống thì chèn khóa mới vào địa chỉ này.

Hàm băm băm lại lần i cho bởi công thức: $h(k) = (h(k)+i)\%M$, trong đó hàm $h(k)$ là hàm băm chính.

Nói dễ hiểu hơn, trong phương pháp này, dây liên tiếp các bước dò tuyến tính là:

- Bước dò thứ 1: $h(k) = k \% M$ (M-Kích thước của bảng băm)
 - Bước dò thứ 2: nếu $h(k)$ đã bị chiếm dụng thì thử dò với $h(k+1)\%M$
 - Bước dò thứ 3: nếu $h(k+1)$ đã bị chiếm dụng thì thử dò với $h(k+2)\%M$
-

- Cứ thế tiếp tục đến dò thứ i : $h(k)=h(k+i-1)\%M$ để tìm chỗ trống, nếu có chỗ trống thì chèn khóa mới vào.

Chèn 76	Chèn 93	Chèn 40	Chèn 47	Chèn 10	Chèn 55
$h(76)=76\%7=6$	$93\%7=2$	$40\%7=5$	$47\%7=5$	$10\%7=3$	$55\%7=6$
0	0	0	0	0	0
1	1	1	1	1	1
2	2 93	2 93	2 93	2 93	2 93
3	3	3	3	3	3
4	4	4	4	4	4
5	5	5 40	5 40	5 40	5 40
6	6 76	6 76	6 76	6 76	6 76
Số bước đếm Probes)	1	1	3	1	3

Hình 2.7. Minh họa quá trình kiến tạo
(chèn lần lượt khóa mới vào) một bảng băm có 7 địa chỉ

+ Tìm một khóa: Trong thao tác này hàm b tìm $h(k)$ sẽ xác định địa chỉ thuộc phạm vi $[0...M-1]$ có thể chứa khóa cần tìm x hay không chứa khóa ấy!

c. Code C mô tả một số thao tác cơ bản trên bảng băm LPM

+ Khởi tri:

+ Khai báo cấu trúc node:

```
struct node
{
    int age, key; //Node của bảng băm gồm các
                   trường: age (tuổi), key (khóa)
    char name[25]; //name (tên)
    int marker; //và trường marker (dánh dấu các
                 nút có xung đột-túc đã bị chiếm dụng)
```

± Chèn khóa mới.

```
void insert(int key, char *name, int age) //Hàm mô  
tả thuật toán chèn trong hash table
```

```

    //với Linear Probing Method
    int hashIndex = key % tableSize; //Thiết lập hàm băm
    if (tableSize == numEle) //Điều kiện báo hiệu hash
        table đầy (Full)
    {
        printf("Hash Table day, khong them phan tu moi
duoc!!!");
        return;
    }
    while (hashTable[hashIndex].marker == 1)//Chừng
        nào có xung đột thì
    {
        hashIndex = (hashIndex + 1)%tableSize;//Băm lại
            (bằng phương pháp dò tuyển tính)
    }
    hashTable[hashIndex].key = key; //Rồi nạp dữ
        liệu vào các trường: key,
    hashTable[hashIndex].age = age; //age,
    strcpy(hashTable[hashIndex].name, name); //name
    hashTable[hashIndex].marker = 1; //Đánh dấu địa chỉ
        xung đột
    numEle++; //Xét nút kế tiếp
    return;
}

```

+ Xóa một khóa

```

void delete(int key)
{
    //Biến count đếm số phần tử trong bảng băm
    int hashIndex = key % tableSize, count = 0, flag = 0;
//Biến flag kiểm tra đúng (1), sai (0)
    if (numEle == 0) //Báo hiệu bảng băm rỗng
    {
        printf("Hash Table rong, khong xoa duoc phan tu
nao!!\n");
        return;
    }

```

```

    }

    while (hashTable[hashIndex].marker != 0 && count <=
tableSize)           //Chừng nào các nút không rỗng
    {
                    //và chưa rà soát hết bằng băm
        if (hashTable[hashIndex].key == key)//thì xác định
                                         //khóa cần xóa
        {

            hashTable[hashIndex].key = 0;      //rồi xóa khóa ấy
            hashTable[hashIndex].marker = -1; //và đánh dấu nút
                                         //đã xóa bởi -1
            hashTable[hashIndex].age = 0; //và xóa dữ liệu ở
                                         //các trường age
            strcpy(hashTable[hashIndex].name, "\0"); //name
            numEle--; //Sau khi xóa, số phần tử của hash table
                       //giảm 1
            flag = 1;//Cập nhật giá trị cờ (biên kiểm tra) flag
            break;
        }

        hashIndex = (hashIndex + 1)%tableSize;//Cập nhật
                                         //băm lại
        count++; //Xét node tiếp theo (cho tới khi số node
}           //bằng kích thước hash table thì dừng!)
        if (flag) //Điều kiện báo hiệu xóa khóa thành công
                  (flag=1)
            printf("Khoa vua nhap da bi xoa!\n");
        else      //Ngược lại, việc ấy bắt thành!
            printf("Nhap khoa khong ton tai trong Hash
Table\n");
        return;
    }
}

+ Tìm một khóa:
void search(int key)           //Hàm mô tả tìm kiếm khóa
{
    int hashIndex = key % tableSize; //trước tiên hàm
                                         //băm xác định địa chỉ (hashIndex) cần chèn
}

```

```

int flag = 0, count = 0;           //Khai báo "cờ" kiểm
                                  tra (đã nói ở trên) và biến đếm các node
if (numEle == 0)                 //Điều kiện báo hiệu hash
                                  table rỗng
{
    printf("Hash Table rỗng, nên không có gì để
tim !");
    return;
}
while (hashTable[hashIndex].marker != 0 && count
<= tableSize) //Chúng nào các nút có dữ liệu
{
    //và chưa xét hết các node của bảng băm, thì lặp
    lại các việc:
    if (hashTable[hashIndex].key == key) //nếu giá
        trị khóa cần tìm bằng giá trị khóa đã lưu ở
        (flag=1; //trường key thì báo hiệu Tìm thấy
         đặt cờ flag=1)
    break; //Kết thúc tìm
}
hashIndex = (hashIndex + 1)%tableSize;//Tiếp tục
                                      dò truyền tính tới các node kế tiếp
count++;
}
if(flag) //Nếu tìm thấy thì in các thông tin sau
          lên màn hình
{
    printf("Da tim thay khoa %d ",hashTable
        [hashIndex].key);
    printf(" o dia chi %d",hashIndex);
}
else printf("Khoa vua nhap khong co trong hash
table!"); // Ngược lại thông báo không tìm thấy
printf("\n\n");
return;
}

```

+ Xem toàn bộ bảng băm:

```

void display()
{
    int i;
    if (numEle == 0)
    {
        printf("In toan bo Hash Table!!:\n");
        return;
    }
    chõ này 5 ký tự trống   8 kt trống   4 kt trống
    printf(" Khoa   Ten   Tuoi   Diachi_bam\n");
    printf("-----\n");
    for (i = 0; i < tableSize; i++)
    {
        if (hashTable[i].marker == 1)
        {
            printf("%-10d", hashTable[i].key); //Các định
            //dạng dữ liệu đoạn program này
            printf("%-13s", hashTable[i].name); //kết hợp
            //với đặt các ký tự trống phù hợp
            printf("%-6d", hashTable[i].age); //trên dòng
            //tiêu đề để khi in kết quả,
            printf("%6d\n", i); //các dữ liệu không bị xô
            //lệch!
        }
    }
    printf("\n");
}

```

+ Thiết kế bảng chọn (Menu):

```

int main()
{
    int key, age, ch;
    char name[25]; clrscr();

```

```
printf("\n\n");
printf("\t\t BANG CHON:\n\n");
printf("Cho biet kich co (tuc la so phan tu) cua
      Hash Table:");
scanf("%d", &tableSize);
hashTable = (struct node *)calloc(tableSize,sizeof
                                  (struct node));
while (1)
{
    printf("1. Chen them\t 2. Xoa khoa\n");
    printf("3. Tim khoa\t4. Xem bang bam\n");
    printf("5. Thoat\n Go [1..5] de chon viec:");
    scanf("%d", &ch);
    switch (ch)
    {
case 1:
        printf("Nhap gia tri khoa:");
        scanf("%d", &key);
        getchar();
        printf("Nhap ten:");
        fgets(name, 25, stdin);
        name[strlen(name) - 1] = '\0';
        printf("Nhap tuoi:"); scanf("%d", &age);
        insert(key, name, age);
        break;
case 2:
        printf("Cho biet gia tri khoa can xoa:");
        scanf("%d", &key);
        delete(key);
        break;
case 3:
        printf("Nhap gia tri khoa can tim:");
```

```

        scanf ("%d", &key);
        search(key);
        break;
    case 4:
        display();
        break;
    case 5:
        exit(0);
    default: printf("Ban da nhap sai muc
                     chon! !\n");
        break;
    }
}
return 0;
}

```

d. Cài đặt tổng thể

Thực chất cài đặt là ráp các modul đã thiết kế trên đây lại với nhau. Công việc này khá dễ dàng nên độc giả tự thực hiện, xem như một bài tự luyện nhỏ.

e. Xét hiệu quả của phương pháp LPM

- Phương pháp băm này chỉ hiệu quả khi băm đều, nghĩa là các nút đã chứa khóa và các nút rỗng xen kẽ nhau. Khi đó ĐPT của việc truy xuất là $O(1)=\text{const}$. Trường hợp tồi nhất là băm không đều hoặc bằng băm đầy, lúc đó các khối (block) đã sử dụng chứa n phần tử và ĐPT của việc truy xuất là $O(n)$.

- Ngoài cách đánh giá trên, người ta còn dùng “hệ số nạp” (Load Factor) để đánh giá hiệu quả của phép dò tuyến tính. Theo định nghĩa, λ xác định bởi tỷ số sau:

$$\lambda = \frac{n}{M}, \text{ trong đó } n - \text{số phần tử, còn } M - \text{kích thước của bảng băm}$$

+ Nếu $\lambda < 1$ thì chắc chắn LPM sẽ tìm được một địa chỉ trống để chèn khóa mới vào.

+ Trong tìm kiếm thì cái giá phải trả cho việc này trên một bảng băm kích thước lớn là:

Tìm kiếm thành công nếu:

$$\lambda = \frac{1}{2} \left(1 + \frac{1}{(1-\lambda)} \right)$$

Tìm kiếm thất bại nếu:

$$\lambda = \frac{1}{2} \left(1 + \frac{1}{(1-\lambda)^2} \right)$$

Với $\lambda > 1/2$ thì LPM suy giảm hiệu quả khá nhanh.

2.2.4. Tạo bảng băm dùng phương pháp dò bậc hai

a. Tổ chức dữ liệu

Trong trường hợp này bảng băm được cài đặt bởi danh sách kề có M nút, mỗi nút là một bản ghi có một trường key chứa giá trị khóa.

b. Một số thao tác cơ bản trên bảng băm dùng phương pháp dò bậc hai (Quadratic Probing Method-QPM)

- Khi khởi tạo bảng băm QPM, mọi trường key của nó đều gán NULL.
- Chèn thêm nút mới vào bảng băm: Trước tiên hàm băm $h(k)$ xác định địa chỉ i thuộc $[0 \dots M-1]$
 - + Nếu không có xung đột tại đó thì thêm nút mới vào địa chỉ này.
 - + Nếu có xung đột tại địa chỉ đó thì hàm băm băm lại lần 1. Ký hiệu hàm băm ở lần này là h_1 .

Hàm h_1 sẽ xác định địa chỉ cách i 1^2 , nếu tại địa chỉ mới này lại có xung đột thì hàm h_1 băm lại lần 2. Ứng với lần này ta có h_2 . Hàm h_2 sẽ xét địa chỉ cách i một khoảng $2^2 \dots$

Quá trình trên tiếp diễn cho tới khi tìm được chỗ trống thì chèn nút mới vào vị trí ấy.

- + Nói rõ hơn dò bậc hai là một dãy liên tiếp các bước dò dưới đây sau khi phát hiện nút đã bị xung đột:

$$(h+1^2)\%M, (h+2^2)\%M, (h+3^2)\%M, (h+4^2)\%M, \dots, (h+i^2)\%M$$

- Tìm kiếm một khóa trong bảng băm: Đầu tiên hàm băm $h(k)$ xét địa chỉ $i=h(k)$, nếu tại đây không có khóa cần tìm thì $h(k)$ tìm tiếp tại địa chỉ cách i một khoảng $1^2, 2^2 \dots$ Quá trình trên tiếp diễn cho tới khi tìm thấy khóa hoặc không tìm thấy khóa khi đã rà soát hết bảng băm.

- Một trong số các hàm băm của phương pháp này có thể chọn như sau:

$$h(k, i) = (h(k) + c_1 i + c_2 i^2) \% M$$

trong đó $c_1, c_2 \neq 0$ là các hằng số. Nếu $c_2=0$ thì dò bậc hai suy biến thành dò tuyến tính.

Thực tế, người ta thường chọn M là số nguyên tố thì băm sẽ đều hơn, ít xung đột hơn.

Ví dụ minh họa về bảng băm QPM có kích thước $M=7$

	chèn 76 6%7=6	chèn 40 40%7=5	chèn 48 48%7=6	chèn 5 5%7=5	chèn 55 55%7=6
0			0 48	0 48	0 48
1			1	1	1
2			2	2 5	2 5
3			3	3	3 55
4			4	4	
5		5 40	5 40	5 40	5 40
6	76	6 76	6 76	6 76	6 76

Số bước dò
(Probes): 1 1 2 3 3

Hình 2.8. Phép chèn với dò bậc hai thành công

	chèn 76 76%7=6	chèn 93 93%7=2	chèn 40 40%7=5	chèn 35 35%7=0	chèn 47 47%7=5
0			0	0 35	0 35
1			1	1	1
2		2 93	2 93	2 93	2 93
3		3	3	3	3 47
4		4	4	4	4
5		5 40	5 40	5 40	5 40
6	76	6 76	6 76	6 76	6 76

Số bước dò
(Probes): 1 1 1 1 ∞

Hình 2.9. Phép chèn với dò bậc hai thất bại

c. Code C mô tả một số thao tác cơ bản trên bảng băm QPM

+ Các khai báo, các thao tác khởi tạo, kiểm tra sự rỗng, kiểm tra sự đầy của bảng băm làm tương tự như các mục trên. Độc giả tự làm như một bài luyện tập nhỏ!

+ Dưới đây chúng tôi chỉ mô tả hai thao tác phức tạp hơn (so với các phương pháp đã tìm hiểu ở trên) trong phương pháp QPM:

+ Hàm mô tả thuật toán chèn:

```
int quadr_prob_insert(int *hash_tab, int k, int *empty)
{
    int h, j=0 //Khai báo kiểu của hàm băm và biến
               //đếm j; gán trị ban đầu cho j=0
    h= k%M;   //Thiết lập hàm băm ban đầu; M: kích thước
               //của bảng băm
    while (j<M) //Chừng nào mà chưa xét hết bảng băm
                  //thì lặp thực thi các lệnh sau:
    {
        if(empty[h]==1) //Nếu vị trí xác định bởi h là rỗng
        { hash_tab(h)=k;//thì chèn khóa k vào địa chỉ ấy
          empty[h]=0; //Hiện tại địa chỉ đó không rỗng nữa
                         //vì đã có khóa mới chèn vào
          return(h); //Hàm chèn trả lại địa chỉ vừa chèn
        }
        j++;           //Tăng biến đếm j lên một
        h=(k+j2)%M; //rồi thực hiện dò bậc hai nút kế tiếp
    }
    return(-1); //Không tìm thấy nút trống thì trả về -1
}
```

+ Hàm mô tả thuật toán tìm khóa:

```
int quadr_prob_find(int *hash_tab, int k, int *empty)
{
    int h, j=0;
    hf=k%M;
```

Khóa cần tìm

```

while (j < M) //Chừng nào chưa rà soát hết bảng băm
    (thì) lặp các lệnh sau:
{
    if ((empty[h] == 0) && (hash_tab[h] == k)) //Nếu địa
        chỉ h không rỗng và tại đó có khóa k
        return (h); //thì tìm thấy ( $\leftrightarrow$  hàm tìm
                      trả về địa chỉ tìm thấy k)
    j++;           //Tăng biến đếm lên một
    h = (k + j2) % M;      //Thực hiện dò bậc hai
}
return -1; //Hàm tìm trả về -1 nếu không tìm thấy
            khóa k trong bảng băm
}

```

e. Xét hiệu quả của phương pháp QPM

Phương pháp này tốt hơn phương pháp dò tuyến tính do phân bổ các khóa đều hơn. Nếu bảng băm chưa đầy thì ĐPT truy xuất là $O(1) = \text{const}$. Nếu bảng băm đầy thì $\text{ĐPT} = O(n)$ (n : số phần tử của bảng băm) do phải thực hiện n so sánh.

Tương tự như ở phương pháp dò tuyến tính, trong phương pháp này người ta còn dùng hệ số nạp λ để đánh giá tính hiệu quả của QPM.

- Nếu M là số nguyên tố và $\lambda \leq 1/2$ thì chắc chắn QPM sẽ tìm được một nút trống sau $M/2$ bước dò hoặc ít hơn.

- Nếu $\lambda > 1/2$ thì QPM có thể thất bại.

f. Một định lý thú vị của phương pháp QPM

+ **Định lý:** Một bảng băm với kích thước M là một số nguyên tố và nếu QPM là khả dụng thì trong $M/2$ bước dò đầu tiên chắc chắn sẽ tìm thấy các nút phân biệt.

+ **Chứng minh:** Ta sẽ chứng minh định lý trên bằng phản chứng!

Giả sử có một nút trống mà nó đã được tìm thấy sau 2 lần dò trong $M/2$ lần lặp đầu tiên.

Ký hiệu i, j là hai lần lặp đó và theo giả thiết $0 \leq i < j \leq M/2$. Chúng ta lại có:

$$(H+i^2)\%M = (H+j^2)\%M \quad (H \text{ là hàm băm})$$

Thế nào cũng chọn được hằng số $k > 0$ sao cho: $H + j^2 = H + i^2 + kM$
(vì $j > i$) $\rightarrow j^2 - i^2 = kM \rightarrow (j-i)(j+i) = kM$ (1)

Vì M là số nguyên tố nên hoặc $(j-i)$ hoặc $(j+i)$ chia hết M .

Trường hợp 1: $(j-i)$ chia hết M .

Từ giả thiết $i < j \leq M/2 \rightarrow (j-i) < M$. Điều này mâu thuẫn.

Trường hợp 2: $j+i$ chia hết M .

Từ giả thiết $i < j \leq M \rightarrow (j+i) < M$. Điều này mâu thuẫn.

Vì vậy không tồn tại 2 bước dò cho cùng một nút trong $M/2$ bước dò đầu tiên.

+ **Hệ quả:** Giá trị nhỏ nhất của i và j để dễ dàng dò được cùng một nút là $i = (M-1)/2$ và $j = (M+1)/2$.

Ta dùng kết quả (1) ở trên để chứng minh hệ quả. Điều mà hệ quả khẳng định sẽ xảy ra khi $k = 1$. Hiển nhiên từ giả thiết của định lý ta có: $0 < i < j \leq M$. Vì $k = 1 \rightarrow (j-i)(j+i) = M$ (2)

Vì M là số nguyên tố nên $(j-i) = M$ hoặc $(j+i) = M$. Từ giả thiết ta lại có: $i < j \leq M \rightarrow (j+i) = M$ và $(j-i) = 1$ (3)

$\rightarrow j = i+1$. Từ (3) và (2) ta có ngay $(j+i) = M \rightarrow (2i+1) = M$. Tức là:

$$i = \frac{M-1}{2} \quad (*)$$

Làm tương tự ta được:

$$j = \frac{M+1}{2} \quad (**) \quad \text{Hệ quả đã được chứng minh!}$$

2.2.5. Tạo bảng băm dùng phương pháp băm kép

a. Chọn hàm băm và tổ chức dữ liệu

Trong phương pháp này người ta dùng hai hàm băm:

- Hàm băm thứ nhất có thể chọn phép chia như ta đã thường làm hoặc dùng phép nhân... Chẳng hạn ta chọn $h1(k) = k \% M$ (k : khóa; M : kích thước của bảng băm và M là số nguyên tố). Hàm băm thứ nhất xác định địa chỉ đầu tiên. Nếu địa chỉ này chưa bị chiếm dụng thì chèn khóa mới vào. Nếu nó đã bị chiếm dụng thì dùng hàm băm thứ hai dưới đây để dò tìm địa chỉ trống (chưa bị chiếm dụng \rightarrow không có đúng độ). Nếu chỗ vừa tìm lại đã bị chiếm dụng thì dò tìm tiếp... cho tới khi tìm được địa chỉ trống thì dừng.

- Một dạng điển hình thường được chọn làm hàm băm thứ hai là:

$$h2(k) = q - k \% q$$

Trong đó q là số nguyên tố và $q < M$. Những giá trị có thể của q thuộc tập $\{1, 2, \dots, q\}$ và $d(k)$ không thể bằng không.

- Với mục đích giải quyết đúng độ, ta sẽ làm một loạt phép dò nhờ biểu thức $(i+j.d(k)) \% M$ với $j = 0, 1, 2, \dots, M-1$.

- Ta dùng mảng $A[]$ lưu các phần tử thuộc bảng băm.

Đầu tiên hàm băm 1 sẽ xác định vị trí i ($i = h(k)$) để thêm khóa mới vào bảng băm.

Giả sử địa chỉ i đã bị chiếm dụng. Khi đó chuỗi các phép thử (dò) sau đây sẽ thực thi:

Thử $A[(i+h2(k)) \% M] = i+h2(k)$ đã bị chiếm dụng

Thử $A[(i+2.h2(k)) \% M] = i+2.h2(k)$ đã bị chiếm dụng

Thử $A[(i+3.h2(k)) \% M] = i+3.h2(k)$ đã bị chiếm dụng

.....

Và cứ thế tiếp tục cho tới khi tìm được địa chỉ trống (để chèn khóa mới vào mà không sinh đúng độ).

b. Ví dụ minh họa giải pháp nêu trên

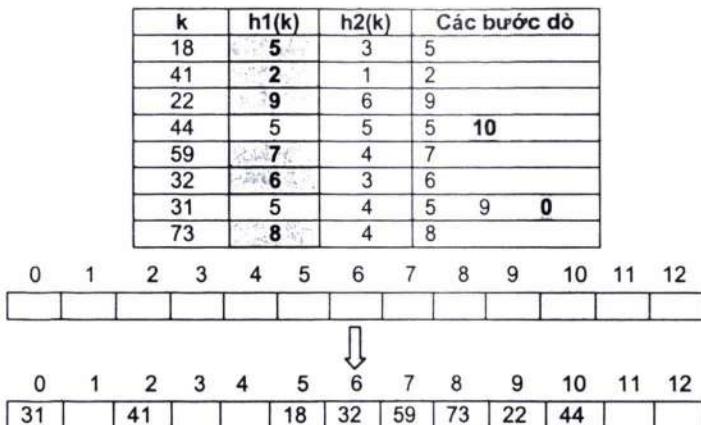
Giả sử bảng băm có kích thước $M = 13$ và ta cần chèn 8 khóa vào bảng băm đó. Ta chọn hàm băm 1 và hàm $d(k)$ như sau:

$$h1(k) = k \% 13;$$

$$h2(k) = q - k \% q \quad \text{Ta chọn } q = 7 \quad (\text{Hãy nhớ: } q \text{ là số nguyên tố và } q < M)$$

$$h2(k) = 7 - k \% 7$$

Các khóa cần chèn vào bảng băm này là: 18, 41, 22, 44, 59, 32, 31, 73.
Quá trình chèn và dò tìm được giải thích ở hình 2.10.



Hình 2.10. Thực hiện băm kép

c. Code C mô tả một vài thao tác cơ bản trên bảng băm dùng phương pháp băm kép (Double Hashing Method-DHM)

+ Khai báo các tiền xử lý chủ chốt:

```
#define khoa_rong -1
#define M 13
#define q 7
#define true 1
#define false 0
```

+ Khai báo cấu trúc nút và bảng băm:

```
typedef struct Node //Khai báo cấu trúc Nút (Node)
{
    int khoa;
} Node;
Node bbam[M]; //Khai báo bảng băm có M địa chỉ
int n; //Khai báo số phần tử hiện có của bảng băm
```

+ Thiết lập các hàm băm:

Hàm băm 1:

```
int h1(int khoa)
```

```

    {
        return (khoa % M);
    }

```

Hàm băm 2:

```

int h2(int khoa)
{
    return (q-(khoa % q)); //Hơi khác với phần lý
                           //thuyết một chút. Nếu bạn
} //muốn có h2 giống như lý thuyết trên đây, cần
   //thêm một biến k nữa!

```

+ Khởi tạo bảng băm:

```

void init()
{
    int i, n=0;
    for(i=0; i<M; i++)
        bbam[i].khoa=khoa_rong;
}

```

+ Kiểm tra xem bảng băm có rỗng không:

```

int empty()
{ return (n==0 ? 1 : 0); } //Hàm băm rỗng thì hàm
                           //empty() trả về 1, ngược lại
                           //là 0

```

+ Kiểm tra bảng băm có đầy không:

```

int full()
{
    return (n==M-1 ? 1 : 0) //Hàm băm đầy thì hàm
                           //full() trả về 1, ngược lại là 0
}

```

+ Tìm kiếm trên bảng băm:

```

int find (int k) //Tìm kiếm khóa k trong bảng băm
{ int i, j;
    i=h1(k); j=h2(k); //Đầu tiên hàm băm h1 xác định
                        //địa chỉ i, còn h2 xác định địa chỉ j
}

```

```

while(bbam[i].khoa=k && bbam[i].khoa!=khoa_rong)
//Nếu khóa cần tìm k bằng khóa lưu ở trường
    i = (i+j) % M; //khóa tại địa chỉ i thì băm lại
                    //bởi h2
    if (bbam[i].khoa==k) //Nếu tìm thấy khóa k
        tại địa chỉ i thì
    return i;           //Hàm find(k) trả về i đó
    else
        return M; //Ngược lại không tìm thấy k
                    trong bảng băm
}
int inse(int k) //Hàm chèn khóa k vào bảng băm
{ int i, j;
    if( full())//Nếu bảng băm đầy thì không chèn được
    { printf("Bang bam day roi!");
        return M;
    }
    if(find(k) < M) //Khóa k đã có trong bảng băm
    { printf("Da co khoa nay trong bang bam.");
        return M;
    }
    i = h1(k); j = h2(k);
    while (bbam[i].khoa!=khoa_rong) //Chừng nào địa
                                    chi i đã bị chiếm dụng
        i = (i + j) % M; //Thì băm lại bởi h2
                            (để tìm chỗ trống)
    bbam[i].khoa=k; //nếu) thấy chỗ trống
                     thì thêm k vào đó
    . . .
    n++; //Tăng số phần tử (khóa) lên 1
    return i; //Cập nhật vị trí đã thêm khóa
}

```

e. Nhận xét về phương pháp DHM

- + So với LPM thì DHM hạn chế được tối đa hiện tượng tụ nhóm (clustering) của dữ liệu (Tụ nhóm nghĩa là có nhiều dữ liệu kề liền nhau “cố tụ” vào một miền trong bảng băm. Hiện tượng này gây lãng phí vùng nhớ (vì bảng băm có nhiều chỗ trống không dùng vào việc gì!) và tốc độ truy xuất giảm đáng kể).

+ DHM do dùng 2 hàm băm khác nhau nên phân bố các khóa ngẫu nhiên và vì thế mà phân bố khóa đồng đều hơn so với các phương pháp khác.

+ Việc tìm kiếm một phần tử trong bảng băm DHM khá khó và phức tạp hơn ở các phương pháp đã xét trước đây.

+ Trong trường hợp tồi nhất, nếu bảng băm có kích thước M thì cần tối đa M^2 bước dò. Nghĩa là thời gian truy xuất tăng khá nhiều nếu M lớn.

+ Nếu bảng băm có ít ($n \ll M$) phần tử (chưa đầy) thì DPT truy xuất là $O(1) = \text{const.}$

Dưới đây là đồ thị so sánh các giải pháp hạn chế đụng độ dùng hai tham số: số bước dò và hệ số nạp λ .

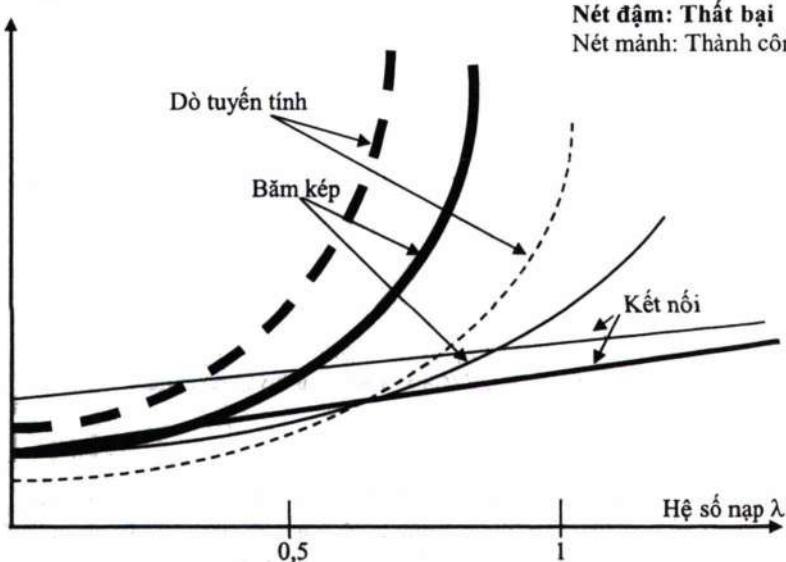
2.3. SO SÁNH CÁC PHƯƠNG PHÁP GIẢI QUYẾT ĐỤNG ĐỘ BẰNG SỐ BƯỚC DÒ

VÀ HỆ SỐ NẠP λ

Dưới đây là đồ thị so sánh:

Số bước dò

Nét đậm: Thất bại
Nét mảnh: Thành công



Hình 2.11. So sánh hiệu quả các phương pháp giải quyết đụng độ dùng số bước dò và hệ số nạp λ

Chương 3

CÂY ĐỎ ĐEN

3.1. MỞ ĐẦU

Khi chèn các dữ liệu đã có trật tự vào cây tìm kiếm nhị phân (Binary Search Tree-BST) thì các tính chất ưu việt của nó biến mất do tính tự cân bằng (self-balancing) của nó bị phá hủy. Để khắc phục nhược điểm này của BST buộc người ta phải nghiên cứu sử dụng một kiểu dữ liệu trừu tượng (ADT) khác. Đó là cây đỏ đen (Red-Black Tree-RBT).

3.2. CÁC KHÁI NIỆM CƠ BẢN VỀ CÂY ĐỎ ĐEN

3.2.1. Định nghĩa

Về thực chất, RBT là cây tìm kiếm nhị phân (BST), song nó được bổ sung thêm các luật dưới đây.

3.2.2. Các đặc điểm (các luật) của RBT

1. Một nút bất kỳ có màu đỏ hoặc đen.
2. Gốc là nút đen. (Tuy nhiên luật này đôi khi bị phớt lờ. Bởi vì nút gốc có thể luôn bị thay đổi từ đỏ sang đen. Nhưng điều ấy cũng chẳng quan trọng gì, bởi vì luật này có hiệu quả rất nhỏ trong việc phân tích cây RBT).
3. Tất cả các lá đều đen. (Còn gọi là các nút NIL (hay nút NULL) và có cùng màu như gốc).
4. Mỗi nút đỏ có 2 nút con đen
5. Mỗi đường đi từ một nút hiện hành đến các hậu duệ của nó đều có cùng một số nút đen.

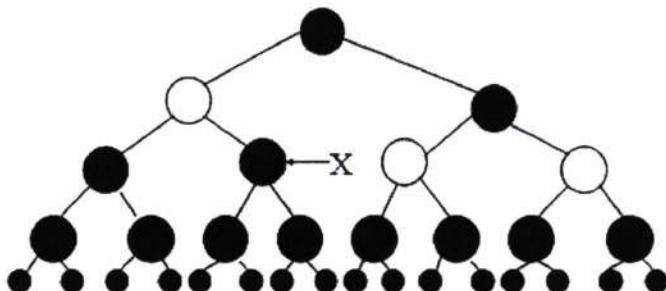
Ví dụ về một cây đỏ đen cho ở hình 3.1.

Chiều cao cây = 4

Chiều cao đen của gốc = 3

Chiều cao đen của nút “X” = 2

Các nút nhỏ dưới cùng là các lá



Quy ước: : nút đen; : nút trắng; N: Nút mới cần chèn vào.

Hình 3.1. Một minh họa về cây đỏ đen

3.2.3. Các tính chất của RBT

Trước khi tìm hiểu các tính chất của RBT, ta cần nêu ra một số định nghĩa dưới đây có liên quan tới các tính chất đó.

a. Định nghĩa chiều cao của RBT

Chiều cao của cây RB là số các cạnh đi từ gốc trên đường đi dài nhất từ gốc đến lá.

b. Định nghĩa chiều cao của nút X

Chiều cao của một nút X là số cạnh trên đường đi dài nhất từ nút X đến lá.

c. Định nghĩa chiều cao đen

Chiều cao đen của một nút X, trên đường đi từ X đến lá là số nút đen trên đường đi tới lá (kể cả nút lá, song không kể nút X).

d. Tính chất 1

Một cây RB với n nút thì chiều cao h của nó nhiều nhất là $2\lg(n + 1)$. Nói khác đi:

$$h \leq 2 * \lg(n+1) \quad (d)$$

Đầu tiên ta sẽ chứng minh 2 bỗ đề dưới đây. Sau đó sẽ chứng minh định lý (d)

e. Tính chất 2 (Bỗ đề a)

Một nút X bất kỳ với chiều cao $h(X)$ có chiều cao đen là $bh(X) \geq h(x)/2$

Chứng minh:

Do luật 4: có ít nhất $h/2$ nút **đỏ** trên đường đi từ nó tới lá, vì vậy có ít nhất $h/2$ nút **đen**.

f. Tính chất 3 (Bỗn đẻ b)

Một cây con với gốc X chứa ít nhất $2^{bh(X)} - 1$ nút.

Chứng minh:

- Ta sẽ chứng minh bằng quy nạp. Cơ sở của quy nạp là $h(X) = 0 \rightarrow X$ là lá nên suy ra $bh(X)=0 \rightarrow$ số các nút là $2^0 - 1 = 0$

- Giả sử giả thiết quy nạp đúng với $h(X) = h-1$

- Ta sẽ chứng minh nó đúng với $h(X) = h$

- Cho $bh(X) = b$, thì một con bất kỳ Y nào đó của X sẽ có:

$bh(Y) = b$ (Nếu con Y là **đỏ**)

$bh(Y) = b - 1$ (Nếu con Y là **đen**)

Dùng giả thiết của quy nạp ta có số nút của mỗi con thuộc X đúng bằng: $2^{bh(x)-1} - 1$

Cây con với gốc X chứa đúng:

$$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2 \cdot (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1 \text{ nút}$$

Đến đây ta dùng kết quả của 2 bỗn đẻ vừa xét trên đây để chứng minh định lý d (Tính chất 1).

Thật vậy ta đã biết:

$$n \geq 2^b - 1 \geq 2^{h/2} - 1 \text{ vì } b \geq h/2$$

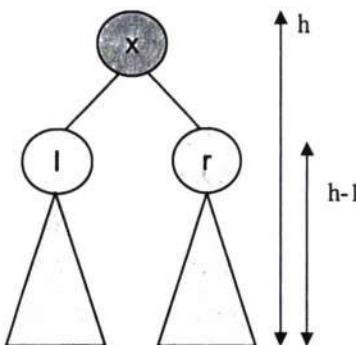
Thêm 1 vào cả 2 vế bất đẳng thức kép trên đây, rồi lấy logarithm cơ số 2 cả hai vế ta được:

$$n+1 \geq 2^b \geq 2^{h/2} \rightarrow \lg(n+1) \geq h/2$$

Vậy: $h \leq 2 * \log(n+1)$ (ở đây log hiểu là log cơ số 2)

Định lý d đã được chứng minh!

Trong chứng minh trên đây, ký hiệu $h=height(Root)$ và $b=bh(Root)$



$$bh(l) \geq bh(x)-1 \quad bh(r) \geq bh(x)-1$$

Hình 3.2. Biểu thị h và $h-1$

3.3. CÁC THAO TÁC CƠ BẢN TRÊN RBT

Trên RBT, khi ta thêm hoặc xóa một khóa tại nút nào đó sẽ dẫn đến sự mất cân bằng của cây và phá hủy các đặc điểm của RBT (RBT là cây tìm kiếm nhị phân đã thêm một số luật). Để tái lập cân bằng và do đó kéo theo tái lập các đặc điểm của BST và RBT ta phải thay đổi quan hệ (về vị trí) giữa một số nút. Nhân đây xin nhắc lại đặc trưng nổi bật của cây tìm kiếm nhị phân (đã tìm hiểu ở Tập 1): Trên các cây con trái mọi khóa đều có giá trị nhỏ hơn giá trị khóa tại nút gốc; trên các cây con phải mọi khóa đều có giá trị lớn hơn giá trị khóa tại gốc.

3.3.1. Phép quay

Việc làm thay đổi quan hệ (về vị trí) giữa một số nút trên RBT để tái lập cân bằng và các thuộc tính của RBT gọi là phép quay (Rotation).

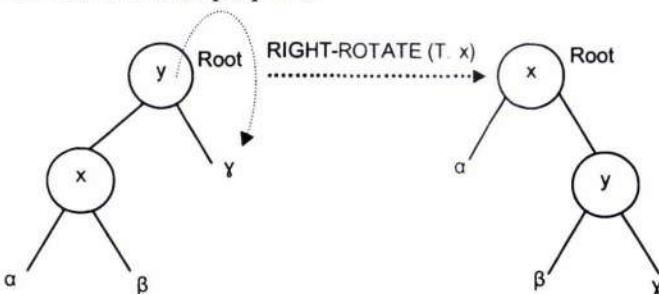
- Ở đây xin lưu ý **độc giả** rằng: Từ định nghĩa rõ ràng và rất đơn giản trên đây, thuật ngữ “Quay” không có nghĩa là quay tròn một nút nào đó như quay chong chóng!

- Trong phép quay, một nút được chọn làm “định” (của phép quay).

a. Phép quay phải (Right- Rotate (RR))

Phép quay phải là phép quay mà nút “định” sẽ chuyển xuống dưới về bên phải “ngồi” vào vị trí nút con phải của nó. Nút con trái chuyển lên trên “ngồi” vào vị trí của nút “định”.

Hình 3.3 minh họa phép RR.



Hình 3.3. Phép quay phải (RR)

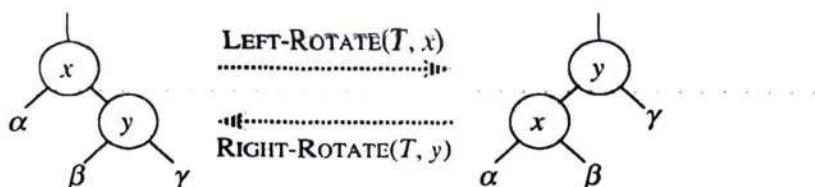
Giả sử ta cần quay phải nút x (tất nhiên x không rỗng)

Phép quay RR phải làm các việc sau:

- Quay phải x trở thành gốc mới của cây con
- Quay phải y trở thành con phải của x
- Con phải của x (β) trở thành con trái của y (β)
- Con trái của x (α) vẫn là con trái của nó

Tương tự như trên ta có phép quay trái dưới đây. Song có sơ đồ và các thao tác ngược lại với RR.

b. Phép quay trái (Left-Rotate (LR))



Hình 3.4. Phép quay trái (LR)

Phép quay LR phải thực hiện các việc sau:

- Quay trái y lên trên trở thành gốc mới của cây con mới
- Quay trái x xuống dưới trở thành cây con trái của y

- Con trái của y (β) thành con phải của x (β)
- Con phải của y (γ) vẫn là con phải của nó

3.3.2. Phép chèn

Phép chèn bắt đầu bằng việc bổ sung một nút như trong cây tìm kiếm nhị phân bình thường và gán cho nó màu đỏ. Ta xem xét việc bảo toàn tính chất đỏ đen từ các nút lân cận với nút mới bổ sung. Thuật ngữ nút chú bác (uncle) sẽ dùng để chỉ nút anh (hoặc em) với nút cha của nút đỏ. Xin nhắc lại một số luật quan trọng:

- Tính chất 3 (Tất cả các lá là đen) giữ nguyên.
- Tính chất 4 (Cả hai con của nút đỏ là đen) nếu bị thay đổi chỉ bởi việc thêm một nút đỏ có thể sửa bằng cách gán màu đen cho một nút đỏ hoặc làm một phép quay.
- Tính chất 5 (Tất cả các đường đi từ gốc tới các lá có cùng một số nút đen) nếu bị thay đổi chỉ bởi việc thêm một nút đỏ có thể sửa bằng cách gán màu đen cho một nút đỏ hoặc làm một phép quay.

Chú ý: Nhãn N sẽ dùng để chỉ nút đang chèn vào, P chỉ nút cha của N, G chỉ ông của N và U chỉ chú bác của N. Nhớ rằng giữa các trường hợp, vai trò và nhãn của các nút có thể thay đổi còn trong cùng một trường hợp thì không.

Mỗi trường hợp được giới thiệu bằng một đoạn mã C. Nút ông và chú bác dễ dàng xác định nhờ các hàm sau:

+ Code C mô tả các nút ông:

```
struct node *grandparent(struct node *N)
{
    if ((N!= NULL) && (N->parent != NULL)) //Nếu N không
                                                rỗng và cha của N không rỗng
        return n->parent->parent; // (thì) N có nút ông
    else
        return NULL; } //Ngược lại không có ông
```

+ Code C xác định các nút ông và chú bác:

```
struct node *uncle(struct node *N)
```

```

{
    struct node *G = grandparent(n);
    if (G == NULL)      // Nếu không có nút ông
        return NULL;    // (thì) cũng không có chú bác
    if (N->parent == G->left) // Nếu cha của N là nút
        ông trái
        return G->right; // (thì) trả về nút ông phải
    else
        return G->left; // Còn thì trả về nút ông trái
}

```

Trường hợp 1: Nút mới thêm N ở tại gốc. Trong trường hợp này, gán lại màu đen cho N, để bảo toàn tính chất 2 (gốc là đen). Vì mới chỉ bổ sung một nút, tính chất 5 được bảo đảm vì mọi đường đi chỉ có cùng một số nút đen.

+ Code mô tả chèn một nút tại gốc (case_1):

```

void insert_case1(struct node *N)
{
    if (N->parent == NULL) // Nếu cha của N rỗng
        N->color = BLACK; // (thì) đặt màu N là Đen
    else
        insert_case2(N); // Ngược lại gọi hàm chèn
                           // trường hợp 2: insert_case2(N)
}

```

Trường hợp 2: Nút cha P của nút mới thêm N là đen (P đen, N đỏ), khi đó tính chất 4 (Cả hai nút con của nút đỏ là đen) không bị vi phạm vì nút mới thêm có hai con là đen. Tính chất 5 cũng không vi phạm vì nút mới thêm là đỏ không ảnh hưởng tới số nút đen trên tất cả các đường đi.

+ Code mô tả chèn case_2 (Cha của nút mới thêm là đen)

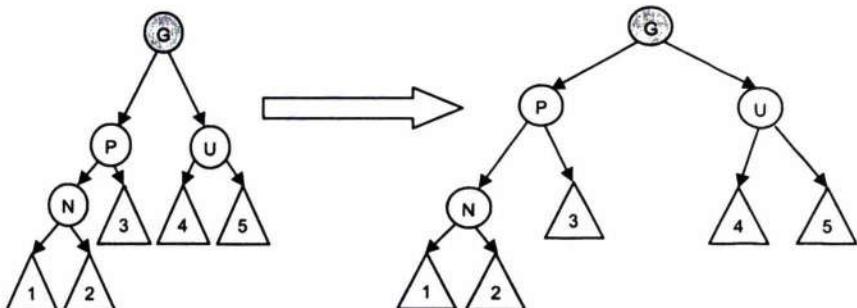
```

void insert_case2(struct node *N)
{
    if (N->parent->color == BLACK) // Nếu cha của N
        là đen
        return; // (thì) RBT vẫn không bị ảnh hưởng gì
    else

```

```
    insert_case3(N); //Ngược lại gọi insert  
                      _case3(N) để chèn  
}
```

Chú ý: Trong trường hợp tiếp theo nếu N có ông là nút G, nếu cha P là đòn và P không ở gốc thì G là đen. Trong trường hợp này N có chú bác là U. (xem hình 3.5). Ta chuyển sang xét trường hợp 3.



Hình 3.5. Chèn trường hợp 3

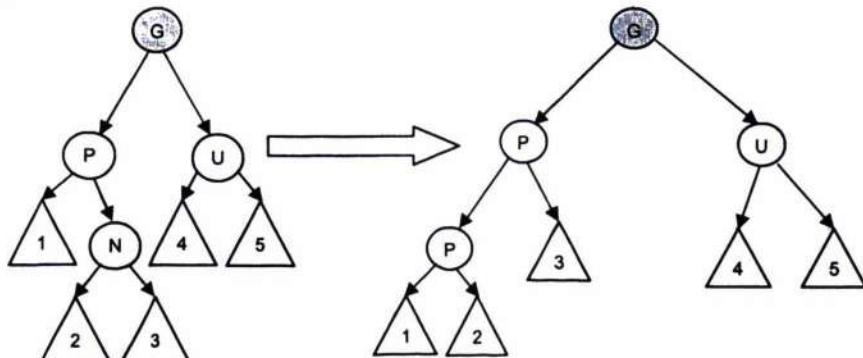
Trường hợp 3: Cả cha P và bác U là đòn, thì đổi cả hai thành đen còn G thành đòn (để bảo toàn tính chất 5) Khi đó nút mới N có cha đen. Vì đường đi bất kỳ đi qua cha và bác của N phải đi qua ông của N nên số các nút đen trên đường đi này không thay đổi. Tuy vậy nút ông G có thể vi phạm tính chất 2 (gốc là đen) hoặc 4 (Cả hai con của nút đòn là nút đen). Để sửa chữa trường hợp này ta gọi một thủ tục để quy trên G từ trường hợp 1.

```

    } //với đối là hàm grandparent() để chèn N
else //Ngược lại gọi hàm insert_case4(N) để chèn N
    insert_case4(N);
}

```

Trường hợp 4: Nút cha P là đỗ nhưng nút chú bác U là đen (hình 3.6), nút mới N là con phải của nút P, và P là con trái của nút G. Trong trường hợp này, thực hiện quay trái chuyển đổi vai trò của nút mới N và nút cha P do đó định dạng lại nút P bằng trường hợp 5 (đổi vai trò N và P) vì tính chất 4 bị vi phạm. Phép quay cũng làm thay đổi một vài đường đi (các đường đi qua cây con nhẵn "1") phải đi qua thêm nút mới N, nhưng vì N là đỗ nên không làm chúng vi phạm tính chất 5.



Hình 3.6. Chèn trường hợp 4

```

void insert_case4(struct node *N)
{
    if (N == N->parent->right && N->parent ==
        grandparent(N)->left) //Nếu 2 điều kiện này đồng thời thỏa mãn
    {
        left_rotate(N->parent); // (thì) quay trái
        // cha của N
        N = N->left;           // (và) đặt N sang trái
    }
    else if (N == N->parent->left && N->parent ==
        grandparent(N)->right) //Ngược lại
    {
        right_rotate(N->parent); // Quay phải cha
        // của N
    }
}

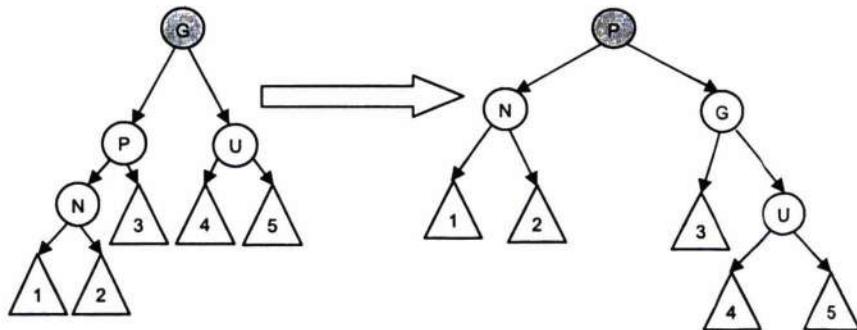
```

```

        N = N->right;      //Đặt N sang phải
    }
    insert_case5(N); //Gọi insert_case5(N) để chèn N
}

```

Trường hợp 5: Nút cha P là đỏ nhưng nút bác U là đen, nút mới N là con trái của nút P và P là con trái của nút ông G. Trong trường hợp này, làm một phép quay phải trên nút ông G; kết quả của phép quay là trong cây mới nút P trở thành cha của cả nút N và nút G. Ban đầu G là đen, còn bây giờ nó là con của P, nên đổi màu của P và G thì cây thỏa mãn tính chất 4. Tính chất 5 không bị vi phạm vì các đường đi qua G trước đây bây giờ đi qua P.



Hình 3.7. Chèn trường hợp 5

```

void insert_case5(struct node *N)
{
    N->parent->color = BLACK; //Đặt màu cha của N (P)
                                //là đen
    grandparent(N)->color = RED; //Đặt màu ông của N (G)
                                    //là đỏ
    if (N == N->parent->left && N->parent ==
        grandparent(N)->left) //Nếu 2 điều kiện này
                                //đồng thời thỏa mãn
    {
        right_rotate(grandparent(N)); //((thì) gọi hàm
                                    //right_rotate() với đối là
    } //hàm grandparent () để chèn N
    else
    {
        //Ngược lại (thì)
    }
}

```

```

    /*Ngược lại ngầm hiểu là N == N->parent->right &&
N->parent == grandparent(N)->right */
    left_rotate(grandparent(n)); //Gọi hàm left_rotate()
                                với đối là
}
                                //grandparent() để chèn N
}

```

3.3.3. Phép xóa

Trong cây tìm kiếm nhị phân bình thường khi xóa một nút có cả hai con (không là lá), ta tìm phần tử lớn nhất trong cây con trái hoặc phần tử nhỏ nhất trong cây con phải, chuyển giá trị của nó vào nút đang muốn xóa (xem Cây tìm kiếm nhị phân - Tập 1). Khi đó chúng ta xóa đi nút đã được sao giá trị, nút này có ít hơn hai con (không là lá). Vì việc sao giá trị không làm mất tính chất đẻ đen nên không cần phải sửa chữa gì cho thao tác này. Việc hiệu chỉnh chỉ đặt ra khi xóa các nút có nhiều nhất một con (không là lá).

Chúng ta sẽ thảo luận về việc xóa một nút có nhiều nhất một con (không là lá).

Nếu ta xóa một nút đẻ, ta có thể chắc chắn rằng con của nó là nút đen. Tất cả các đường đi đi qua nút bị xóa chỉ đơn giản bớt đi một nút đẻ do đó luật 5 không bị vi phạm. Ngoài ra, cả nút cha và nút con của nút bị xóa đều là nút đen, do đó tính chất 3 và 4 cũng không bị vi phạm. Một trường hợp đơn giản khác là khi xóa một nút đen chỉ có một con là nút đẻ. Khi xóa nút đó các luật 4 và 5 bị phá vỡ, nhưng nếu gán lại màu cho nút con là đen thì các luật đó lại được khôi phục.

Trường hợp phức tạp xảy ra khi cả nút bị xóa và nút con của nó đều là đen. Chúng ta sẽ bắt đầu bằng việc thay nút bị xóa bằng nút con của nó (tên khác: nút thế mạng). Chúng ta ký hiệu nút con này (trong vị trí mới của nó là N và anh em với nó (con khác của nút cha mới) là S. Tiếp theo ta vẫn dùng P chỉ cha mới của N, S_L chỉ con trái của S và S_R chỉ con phải của S (chúng tồn tại vì S không thể là lá).

Code mô tả thuật toán xác định nút anh em của N:

```

struct node *sibling(struct node *N)
{
    if (N == N->parent->left) //Nếu cha của N ở
                                bên trái

```

```

        return N->parent->right;      // (thì) anh, em của N
                                         ở bên phải
    else
        return N->parent->left;     // Ngược lại anh, em
                                         của N ở bên trái
}

```

Chú ý: Nếu nút bị xóa N không có con khác lá, dễ dàng thấy rằng các tính chất được thỏa mãn. Còn nếu N là một lá, có thể hiệu chỉnh lược đồ (hoặc code) để trong tất cả các trường hợp các tính chất được thỏa mãn. Trước mỗi bước ta có thể dùng hàm (function) `replace_node` thay thế nút con `child` vào vị trí của nút bị xóa trên cây.

```

void delete_one_child(struct node *N)           // Giả thiết:
                                                 N có ít nhất một nút con trái
{
    // Hàm này xóa nút thê mạng của N
    struct node *child = is_leaf(N->right) ? N->left :
N->right; // Nếu N là lá (trái hoặc phải)
    replace_node(N, child);                      // (thì) gọi hàm
                                                 replace_node() thay N bằng nút con của nó
    if (N->color == BLACK)                      // Nếu N có màu đen (thì)
    {
        if (child->color == RED)               // Nếu con của nó
                                                 có màu đỏ
            child->color = BLACK;             // (thì) đặt lại màu
                                                 con của N là đen
        else                                // Ngược lại
            delete_casel(child);           // Gọi delete_casel()
                                                 để xóa nút thê mạng
    }
    free(N);                                  // Xóa nút N
}

```

Nếu cả N và gốc ban đầu của nó là đen thì sau khi xóa các đường qua N giảm bớt một nút đen. Do đó vi phạm luật 5, cây cần phải cân bằng lại. Chúng ta xét các trường hợp sau:

Trường hợp 1: N là gốc mới. Trong trường hợp này chúng ta dùng lại. Ta đã giải phóng một nút đen khỏi mọi đường đi và gốc mới lại là đen. Không luật nào bị vi phạm.

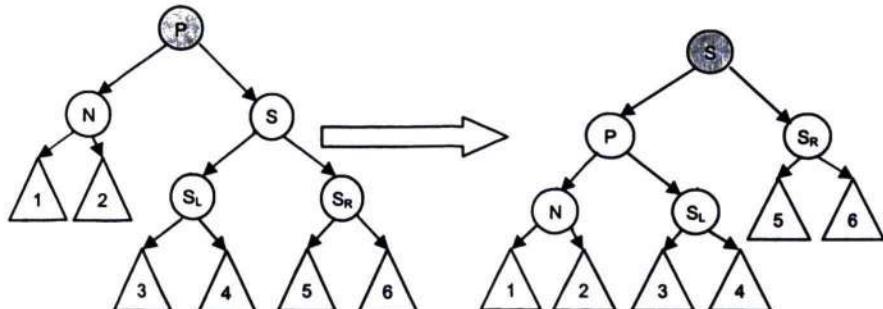
```

void delete_case1(struct node *N)
{
    if (N->parent == NULL)           //Nếu N là gốc
        return; // (thì) cây đỏ đen không thay đổi gì
    else                         //Ngược lại
        delete_case2(N); //Gọi hàm delete_case2() xóa N
}

```

Chú ý: Trong các trường hợp 2, 5 và 6 dưới đây, ta quy ước N là con trái của cha P. Nếu nó là con phải thì con trái và phải sẽ tráo đổi cho nhau.

Trường hợp 2: S là đỏ. Trong trường hợp này tráo đổi màu của P và S và sau đó quay trái tại P, nó sẽ làm cho S trở thành nút ông của N. Chú ý rằng P có màu đen và có một con màu đỏ. Tất cả các đường đi có số các nút đen giống nhau, bây giờ N có một anh em màu đen (S_L) và cha (P) màu đỏ.



Hình 3.8. Xóa nút trường hợp 2

```

void delete_case2(struct node *N)
{
    if (sibling(N)->color == RED) //Nếu anh, em của N
                                    có màu đỏ
    {
        N->parent->color = RED;      // (thì) đặt màu
                                        cha của nó là màu đỏ
        sibling(N)->color = BLACK; //còn anh, em của N
                                    có màu đen
        if (N == N->parent->left)   //Nếu N ở bên
                                        trái cha (thì)
            rotate_left(N->parent); //quay trái cha của N
    }
}

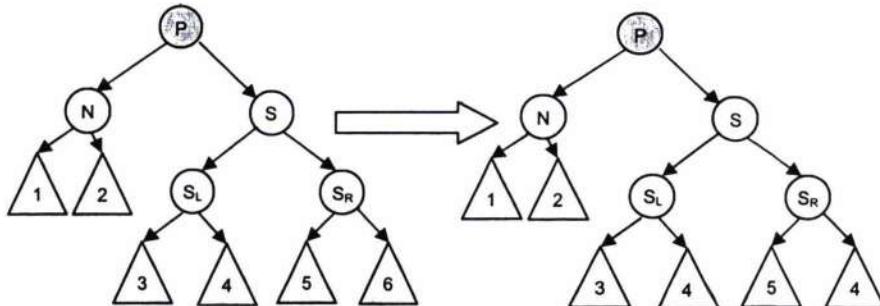
```

```

else
    rotate_right(N->parent); //Ngược lại quay phải cha của N
}
delete_case3(N);           //Gọi delete_case3() xóa N
}

```

Trường hợp 3: P, S và các con của S là đen. Trong trường hợp này, chúng ta gán lại cho S màu đỏ. Kết quả là mọi đường đi qua S, (tất nhiên chúng không qua N), có ít hơn một nút đen. Vì việc xóa đi cha trước đây của N làm tất cả các đường đi qua N bớt đi một nút đen, nên chúng bằng nhau. Tuy nhiên tất cả các đường đi qua P bây giờ có ít hơn một nút đen so với các đường không qua P, do đó tính chất 5 (tất cả các đường đi từ gốc tới các nút lá có cùng số nút đen) sẽ bị vi phạm. Để hiệu chỉnh hiện tượng này ta lại tái cân bằng tại P, như trường hợp 1.



Hình 3.9. Xóa nút trường hợp 3

```

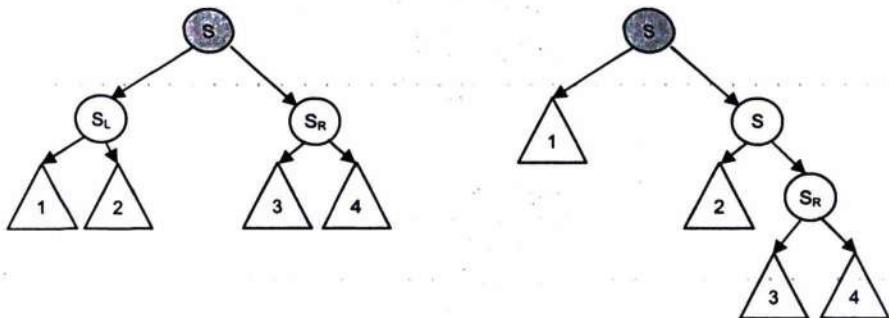
void delete_case3(struct node *N)
{
    //Nếu 4 điều kiện dưới đây đồng thời thỏa mãn (thì)
    if (N->parent->color == BLACK && sibling(N)->color == BLACK
        && sibling(N)->left->color == BLACK
        && sibling(N)->right->color == BLACK)
    {
        sibling(N)->color = RED; //Đặt màu anh, em của N
                                   //là đỏ
        delete_case1(N->parent); //Gọi delete_case1() xóa
                                   //cha của N
    }
    else
        delete_case4(N); //Ngược lại gọi delete_case4()
                           //xóa N
}

```

Trường hợp 4: S và các con của S là đen nhưng P là đỏ (Độc giả tự vẽ sơ đồ cho trường hợp đơn giản này!). Trong trường hợp 4, ta đổi ngược màu của S và P. Điều này không ảnh hưởng tới số nút đen trên các đường đi không qua N, nhưng thêm một nút đen trên các đường đi qua N, thay cho nút đen đã bị xóa trên các đường này.

```
void delete_case4(struct node *N)
{
    //Nếu 4 điều kiện dưới đây đồng thời thỏa mãn (thì)
    if (N->parent->color == RED && sibling(N)->color == BLACK && sibling(N)->left->color == BLACK && sibling(N)->right->color == BLACK)
    {
        sibling(N)->color = RED; //Đặt màu anh, em của N là đỏ
        N->parent->color = BLACK; //Đặt màu cha của N là đen
    }
    else //Ngược lại
        delete_case5(N); //Gọi delete_case5() xóa N
}
```

Trường hợp 5: S là đen, con trái của S (S_L) là đỏ, con phải của S (S_R) đen. Trong trường hợp này, quay phải tại S. Khi đó S_L trở thành cha của S, S_R trở thành con của S (và là cháu của S_L). Chúng ta đổi màu S_L thành đen, S màu đỏ, S_R màu đen. Tất cả các đường đi có số nút đen như nhau.



Hình 3.10. Xóa nút trường hợp 5

```

void delete_case5(struct node *N) //Hàm xóa nút trường
    hợp 5
{
    //Giải thích các lệnh trong hàm này tương tự các
    trường hợp trên
    if (N == N->parent->left && sibling(N)->color == BLACK && sibling(N)->left->color == RED && sibling(N)->right->color == BLACK)
    {
        sibling(N)->color = RED;
        sibling(N)->left->color = BLACK;
        rotate_right(sibling(N));
    }
    else if (N == N->parent->right && sibling(N)->color == BLACK && sibling(N)->right->color == RED && sibling(N)->left->color == BLACK)
    {
        sibling(N)->color = RED;
        sibling(N)->right->color = BLACK;
        rotate_left(sibling(N));
    }
    delete_case6(N);
}

```

Trường hợp 6: (Lưu ý trong trường hợp 6, ta đặt lại nút anh em mới của N là S)

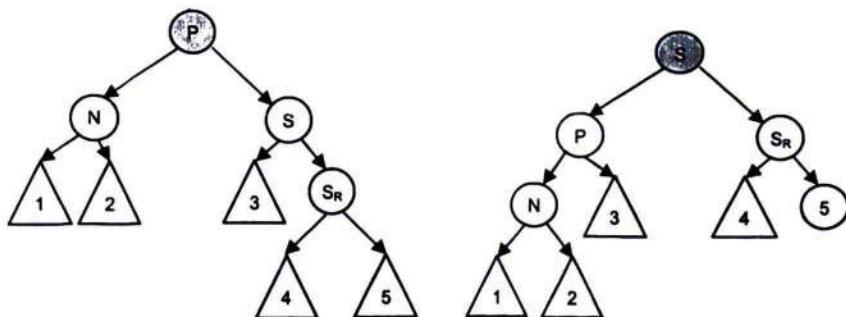
S là đen, con phải của S là đỏ và N là con trái của nút cha P. Trong trường hợp này chúng ta quay trái tại P, khi đó S trở thành cha của P và con phải của S là Sr. Chúng ta hoán đổi màu của P và S và gán cho con phải của S màu đen. Cây con giữ nguyên màu của gốc do đó tính chất 4 (Cả hai con của nút đỏ là đen) và tính chất 5 không bị vi phạm trong cây con này. Tuy nhiên, N bây giờ có thêm một nút đen tiền bối: hoặc P mới bị tô đen, hoặc S là nút ông của nó trở thành đen. Như vậy các đường đi qua N có thêm một nút đen.

Trong lúc đó, với một đường đi không đi qua N, có hai khả năng:

- Đi qua nút anh em của N. Khi đó cả trước và sau khi quay nó phải đi qua S và P, khi thay đổi màu sắc hai nút này đã tráo đổi màu cho nhau. Như vậy đường đi này không bị thay đổi số nút đen.

- Đi qua nút bác của N, là con phải của S. Khi đó trước khi quay nó đi qua cha của S và con phải của S, nhưng sau khi quay nó chỉ đi qua nút S và con phải (SR) của S; Lúc này S đã nhận màu cũ của cha P, còn con phải của S đã đổi màu từ đỏ thành đen. Kết quả là số các nút đen trên đường đi này không thay đổi.

Như vậy, số các nút đen trên các đường đi là không thay đổi. Do đó các tính chất 4 và 5 đã được khôi phục. Nút trắng trong hình vẽ có thể là đỏ hoặc đen, nhưng phải ghi lại trước và sau khi thay đổi.



Hình 3.11. Xóa nút trường hợp 6

```

void delete_case6(struct node *N)
{
    sibling(N)->color = N->parent->color; //Đổi màu
                                              //giữa anh, em của N và cha của N
    N->parent->color = BLACK; //Bây giờ đặt màu cho
                               //cha N là đen
    if (N == N->parent->left) //Nếu N ở bên trái (thì)
    {
        /* Ở đây, sibling(n)->right->color == RED */
        sibling(N)->right->color = BLACK; //Đặt màu cho anh, em
                                              //bên phải N là đen
        rotate_left(N->parent); //Quay trái cha của N
    }
    Else                                //Ngược lại
    {

```

```

    /* Ở đây sibling(N)->left->color == RED */ sibling(N)-
>left->color = BLACK;      //Đặt màu cho anh, em bên trái N
là đen
    rotate_right(N->parent); //Quay phải cha của N
}

```

Đề ý rằng các hàm này có lời gọi đệ quy. Hơn nữa lời gọi không đệ quy sẽ được gọi sau một phép quay, do đó số lần thực hiện các phép quay là không đổi (không quá 3).

3.4. CÀI ĐẶT TRÊN C++

Về cơ bản cài đặt sau đây bám sát ý tưởng chủ đạo đã phân tích ở phần trên. Song về chi tiết có thể tồn tại một số điểm khác vì lý do phải tuân thủ kỹ thuật lập trình hướng đối tượng với C++.

```

#include<iostream.h>
#include<stdio.h>
#include<string.h>
#include<algorithm.h>
#include<math.h>
#include<vector.h>
#include<stdlib.h>
#include<assert.h>

#define INDENT_STEP 4

enum color { RED, BLACK };
typedef struct rbtree_node //Khai báo cấu trúc của
{
    enum color color;
    void *key;
    void *value;
    rbtree_node *left, *right, *parent;
}*node;

```

```
typedef struct rbtree_t //Khai báo cấu trúc cây dò đen
{
    node root;
}*rbtree;

//Khai báo lớp cây dò đen và các nguyên mẫu

class RBTree
{
public:
    typedef int (*compare_func)(void* left, void* right);
    rbtree rbtree_create();
    void* rbtree_lookup(rbtree t, void*, compare_func
                        compare);
    void rbtree_insert(rbtree t, void*, void* ,
                      compare_func compare);
    void rbtree_delete(rbtree t, void*, compare_func
                      compare);
    node grandparent(node n);
    node sibling(node n);
    node uncle(node n);
    void verify_properties(rbtree t);
    void verify_property_1(node root);
    void verify_property_2(node root);
    color node_color(node n);
    void verify_property_4(node root);
    void verify_property_5(node root);
    void verify_property_5_helper(node n, int , int*);
    node new_node(void* key, void*, color, node, node);
    node lookup_node(rbtree t, void*, compare_func
                     compare);
    void rotate_left(rbtree t, node n);
    void rotate_right(rbtree t, node n);
```

```
void replace_node(rbtree t, node oldn, node newn);
void insert_case1(rbtree t, node n);
void insert_case2(rbtree t, node n);
void insert_case3(rbtree t, node n);
void insert_case4(rbtree t, node n);
void insert_case5(rbtree t, node n);
node maximum_node(node root);
void delete_case1(rbtree t, node n);
void delete_case2(rbtree t, node n);
void delete_case3(rbtree t, node n);
void delete_case4(rbtree t, node n);
void delete_case5(rbtree t, node n);
void delete_case6(rbtree t, node n);
};

Node RBTree::grandparent(node n) //Trả về nút ông của
                                 nút đang xét
{
    assert (n != NULL);
    assert (n->parent != NULL);
    assert (n->parent->parent != NULL);
    return n->parent->parent;
}

node RBTree::sibling(node n) //Trả về các nút anh em
                           của nút đang xét
{
    assert (n != NULL);
    assert (n->parent != NULL);
    if (n == n->parent->left)
        return n->parent->right;
    else
        return n->parent->left;
```

```
}

node RBTree::uncle(node n)           //Trả về nút chú bác
{                                     của nút đang xét
    assert (n != NULL);
    assert (n->parent != NULL);
    assert (n->parent->parent != NULL);
    return sibling(n->parent);
}

void RBTree::verify_properties(rbtree t)
//Kiểm tra một vài thuộc tính của RBT
{
    verify_property_1 (t->root);
    verify_property_2 (t->root);
    verify_property_4 (t->root);
    verify_property_5 (t->root);
}

void RBTree::verify_property_1(node n)   //Kiểm tra
                                         thuộc tính 1
{
    assert (node_color(n) == RED || node_color(n) ==
BLACK);
    if (n == NULL)
        return;
    verify_property_1(n->left);
    verify_property_1(n->right);
}

void RBTree::verify_property_2(node root) //Kiểm tra
                                         thuộc tính 2
{
    assert (node_color(root) == BLACK);
}
```

```

color RBTree::node_color(node n)           //Trả về màu của
                                           nút đang xét
{
    return n == NULL ? BLACK : n->color;
}

void RBTree::verify_property_4(node n)      //Kiểm tra
                                           thuộc tính 4
{
    if (node_color(n) == RED)
    {
        assert (node_color(n->left) == BLACK);
        assert (node_color(n->right) == BLACK);
        assert (node_color(n->parent) == BLACK);
    }
    if (n == NULL)
        return;
    verify_property_4(n->left);
    verify_property_4(n->right);
}

void RBTree::verify_property_5(node root)    //Kiểm tra
                                           thuộc tính 5
{
    int black_count_path = -1;
    verify_property_5_helper(root, 0,&black_count_path);
}

/* Dưới đây là hàm hỗ trợ kiểm tra thuộc tính 5, nó
cập nhật biến đếm nút đen và cập nhật biến đếm đường đi
đen (đường đi chỉ gồm các nút đen) */
void RBTree::verify_property_5_helper(node n, int
black_count, int* path_black_count)
{
    if (node_color(n) == BLACK)

```

```
{  
    black_count++;  
}  
if (n == NULL)  
{  
    if (*path_black_count == -1)  
    {  
        *path_black_count = black_count;  
    }  
    else  
    {  
        assert (black_count == *path_black_count);  
    }  
    return;  
}  
verify_property_5_helper(n->left, black_count,  
                        path_black_count);  
verify_property_5_helper(n->right, black_count,  
                        path_black_count);  
}  
  
rbtree RBTree::rbtree_create()           //Tạo cây RB  
{  
    rbtree t = new rbtree_t;  
    t->root = NULL;  
    verify_properties(t);  
    return t;  
}  
//Dưới đây là hàm tạo nút mới cho cây RB  
node RBTree::new_node(void* k, void* v, color  
n_color, node left, node right)
```

```
{  
    node result = new rbtree_node;  
    result->key = k;  
    result->value = v;  
    result->color = n_color;  
    result->left = left;  
    result->right = right;  
    if (left != NULL)  
        left->parent = result;  
    if (right != NULL)  
        right->parent = result;  
    result->parent = NULL;  
    return result;  
}  
  
node RBTree::lookup_node(rbtree t, void* key,  
compare_func compare) //Tim một nút  
{  
    node n = t->root;  
    while (n != NULL)  
    {  
        int comp_result = compare(key, n->key);  
        if (comp_result == 0)  
        {  
            return n;  
        }  
        else if (comp_result < 0)  
        {  
            n = n->left;  
        }  
        else  
        {  
            assert(comp_result > 0);  
        }  
    }  
}
```

```
n = n->right;
}
}
return n;
}

void* RBTree::rbtree_lookup(rbtree t, void* key,
compare_func compare) //Tim kiém trên RBT
{
    node n = lookup_node(t, key, compare);
    return n == NULL ? NULL : n->value;
}

void RBTree::rotate_left(rbtree t, node n)
//Thuật toán quay trái (Left-Rotate)
{
    node r = n->right;
    replace_node(t, n, r);
    n->right = r->left;
    if (r->left != NULL)
    {
        r->left->parent = n;
    }
    r->left = n;
    n->parent = r;
}

void RBTree::rotate_right(rbtree t, node n) //Thuật
    toán quay phải (Right-Rotate)

{
    node L = n->left;
    replace_node(t, n, L);
    n->left = L->right;
```

```

if (L->right != NULL)
{
    L->right->parent = n;
}
L->right = n;
n->parent = L;
}

void RBTree::replace_node(rbtree t, node oldn, node newn) //Thay thế một nút
{
    if (oldn->parent == NULL)
    {
        t->root = newn;
    }
    else
    {
        if (oldn == oldn->parent->left)
            oldn->parent->left = newn;
        else
            oldn->parent->right = newn;
    }
    if (newn != NULL)
    {
        newn->parent = oldn->parent;
    }
}

//Thuật toán chèn
void RBTree::rbtree_insert(rbtree t, void* key, void*
value, compare_func compare)
{
    node inserted_node = new_node(key, value, RED, NULL,
NULL);

```

```
if (t->root == NULL)
{
    t->root = inserted_node;
}
else
{
    node n = t->root;
    while (1)
    {
        int comp_result = compare(key, n->key);
        if (comp_result == 0)
        {
            n->value = value;
            return;
        }
        else if (comp_result < 0)
        {
            if (n->left == NULL)
            {
                n->left = inserted_node;
                break;
            }
        }
        else
        {
            n = n->left;
        }
    }
}
assert (comp_result > 0);
if (n->right == NULL)
{
    n->right = inserted_node;
```

```
        break;
    }
    else
    {
        n = n->right;
    }
}
inserted_node->parent = n;
}
insert_case1(t, inserted_node);
verify_properties(t);
}

void RBTree::insert_case1(rbtree t, node n)
//Thuật toán chèn của trường hợp 1
{
    if (n->parent == NULL)
        n->color = BLACK;
    else
        insert_case2(t, n);
}

void RBTree::insert_case2(rbtree t, node n) //Thuật
                                             toán chèn của trường hợp 2
{
    if (node_color(n->parent) == BLACK)
        return;
    else
        insert_case3(t, n);
}
```

```
void RBTree::insert_case3(rbtree t, node n)
//Thuật toán chèn của trường hợp 3
{
    if (node_color(uncle(n)) == RED)
    {
        n->parent->color = BLACK;
        uncle(n)->color = BLACK;
        grandparent(n)->color = RED;
        insert_casel(t, grandparent(n));
    }
    else
    {
        insert_case4(t, n);
    }
}

void RBTree::insert_case4(rbtree t, node n)
//Thuật toán chèn của trường hợp 4
{
    if (n == n->parent->right && n->parent ==
        grandparent(n)->left)
        (rotate_left(t, n->parent);
        n = n->left;
    }
    else if (n == n->parent->left && n->parent ==
              grandparent(n)->right)
    {
        rotate_right(t, n->parent);
        n = n->right;
    }
    insert_case5(t, n);
}
```

```
void RBTree::insert_case5(rbtree t, node n)
//Thuật toán chèn của trường hợp 5
{
    n->parent->color = BLACK;
    grandparent(n)->color = RED;
    if (n == n->parent->left && n->parent ==
        grandparent(n)->left)
    {
        rotate_right(t, grandparent(n));
    }
    else
    {
        assert (n == n->parent->right && n->parent
                == grandparent(n)->right);
        rotate_left(t, grandparent(n));
    }
}

//Xóa một nút trên cây RB
void RBTree::rbtree_delete(rbtree t, void* key,
compare_func compare)
{
    node child;
    node n = lookup_node(t, key, compare);
    if (n == NULL)
        return;
    if (n->left != NULL && n->right != NULL)
    {
        node pred = maximum_node(n->left);
        n->key = pred->key;
        n->value = pred->value;
        n = pred;
    }
}
```



```

    {
        if (node_color(sibling(n)) == RED)
        {
            n->parent->color = RED;
            sibling(n)->color = BLACK;
            if (n == n->parent->left)
                rotate_left(t, n->parent);
            else
                rotate_right(t, n->parent);
        }
        delete_case3(t, n);
    }

void RBTree::delete_case3(rbtree t, node n)      //Thuật
                                                 toán xóa với trường hợp 3
{
    if (node_color(n->parent) == BLACK &&
node_color(sibling(n)) == BLACK &&
        node_color(sibling(n)->left) == BLACK &&
node_color(sibling(n)->right) == BLACK)
    {
        sibling(n)->color = RED;
        delete_casel(t, n->parent);
    }
    else
        delete_case4(t, n);
}

void RBTree::delete_case4(rbtree t, node n)      //Thuật
                                                 toán xóa với trường hợp 4
{
    if (node_color(n->parent) == RED &&
node_color(sibling(n)) == BLACK &&
        node_color(sibling(n)->left) == BLACK &&
node_color(sibling(n)->right) == BLACK)
}

```

```
{  
    sibling(n)->color = RED;  
    n->parent->color = BLACK;  
}  
else  
    delete_case5(t, n);  
}  
  
void RBTree::delete_case5(rbtree t, node n)      //Thuật  
          toán xóa với trường hợp 5  
{  
    if (n == n->parent->left && node_color(sibling(n))  
        == BLACK &&  
        node_color(sibling(n)->left) == RED &&  
        node_color(sibling(n)->right) == BLACK)  
    {  
        sibling(n)->color = RED;  
        sibling(n)->left->color = BLACK;  
        rotate_right(t, sibling(n));  
    }  
    else if (n == n->parent->right &&  
             node_color(sibling(n)) == BLACK &&  
             node_color(sibling(n)->right) == RED &&  
             node_color(sibling(n)->left) == BLACK)  
    {  
        sibling(n)->color = RED;  
        sibling(n)->right->color = BLACK;  
        rotate_left(t, sibling(n));  
    }  
    delete_case6(t, n);  
}
```

```
void RBTree::delete_case6(rbtree t, node n)
//Xóa nút ở trường hợp 6
{
    sibling(n)->color = node_color(n->parent);
    n->parent->color = BLACK;
    if (n == n->parent->left)

    {
        assert (node_color(sibling(n)->right) == RED);
        sibling(n)->right->color = BLACK;
        rotate_left(t, n->parent);
    }
    else
    {
        assert (node_color(sibling(n)->left) == RED);
        sibling(n)->left->color = BLACK;
        rotate_right(t, n->parent);
    }
}

int compare_int(void* leftp, void* rightp)
//Hàm so sánh giá trị khóa của 2 nút
{
    int left = (int)leftp;
    int right = (int)rightp;
    if (left < right)
        return -1;
    else if (left > right)
        return 1;
    else
    {
        assert (left == right);
        return 0;
    }
}
```

```
    }

}

void print_tree_helper(node n, int indent) //Hàm phụ
                                         //trợ việc in cây
{
    int i;
    if (n == NULL)
    {
        fputs("<Cay rong>", stdout);
        return;
    }
    if (n->right != NULL)
    {
        print_tree_helper(n->right, indent +
                           INDENT_STEP);
    }
    for(i = 0; i < indent; i++)
        fputs(" ", stdout);
    if (n->color == BLACK)
        cout<<(int)n->key<<endl;
    else
        cout<<"<"<<(int)n->key<<">"<<endl;
    if (n->left != NULL)
    {
        print_tree_helper(n->left, indent +
                           INDENT_STEP);
    }
}
void print_tree(rbtree t)
//Hàm in cây lên màn hình
{
    print_tree_helper(t->root, 0);
    puts(" ");
}
```

```

}

int main()           //Hàm chính thiết kế Menu
{
    int i;
    RBTree rbt;
    rbtree t = rbt.rbtree_create();
    for (i = 0; i < 12; i++)
    {
        int x = rand() % 10;
        int y = rand() % 10;
        print_tree(t);
        cout<<"Chen khoa moi. "<<x<<" ->
"<<y<<endl<<endl;
        rbt.rbtree_insert(t, (void*)x, (void*)y,
                           compare_int);
        assert(rbt.rbtree_lookup(t, (void*)x,
                               compare_int) == (void*)y);
    }
    for (i = 0; i < 15; i++)
    {
        int x = rand() % 10;
        print_tree(t);
        cout<<" Xoa mot khoa. "<<x<<endl<<endl;
        rbt.rbtree_delete(t, (void*)x, compare_int);
    }
    return 0;
}

```

3.5. ĐÁNH GIÁ HIỆU QUẢ CỦA RBT

Theo William H. Ford và William R. Topp, đồng tác giả của tài liệu dẫn [6], hiệu quả của cây RB là:

Phép tìm kiếm:	O(h)	h: Chiều cao của cây RB
Phép tìm nút tiền bối:	O(h)	
Phép tìm nút hậu bối:	O(h)	
Phép tìm khóa lớn nhất:	O(h)	
Phép tìm khóa nhỏ nhất:	O(h)	
Phép chèn:	O(h)	
Phép xóa:	O(h)	

Chương 4

CÂY 2-3-4 VÀ B-CÂY

4.1. CÂY 2-3-4

4.1.1. Mở đầu

Với cây đòn đen ta không thể lưu được nhiều mục dữ liệu tại mỗi nút. Trong thực tế có những nhu cầu cần lưu nhiều mục dữ liệu ở cùng một nút của cây. Cây 2-3-4 đáp ứng được yêu cầu này.

Trong khoa học máy tính, cây 2-3-4 là cây nhiều nhánh mà mỗi nút của nó có thể có đến bốn nút con và ba mục dữ liệu. Cây 2-3-4 là cây cân bằng giống như cây đòn đen, tuy nhiên ít hiệu quả hơn nhưng ngược lại dễ lập trình hơn.

Các số 2, 3, 4 trong cụm từ cây 2-3-4 mang ý nghĩa là có 2, 3 hoặc 4 liên kết đến các nút con có thể có được trong một nút cho trước.

Tất cả các nút lá đều không có nút con, nhưng có thể chứa 1, 2 hoặc 3 mục dữ liệu, không có nút rỗng (khác với cây đòn đen). Một cây 2-3-4 có thể có đến bốn cây con, nên được gọi là cây nhiều nhánh bậc 4.

Trong cây 2-3-4 mỗi nút có ít nhất là hai liên kết, trừ nút lá (nút không có nút con).

4.1.2. Tô chức cây 2-3-4

a. Các khóa trong một nút

Trong một nút lá hoặc nút trong (những nút khác lá gọi là nút trong) có thể có 2-3-4 khóa đại diện cho mục dữ liệu. Các khóa trong mỗi nút được sắp xếp theo thứ tự tăng dần. Tất nhiên các nút lá không có con. Còn với các nút không phải là lá, trong từng trường hợp có số nút con như sau:

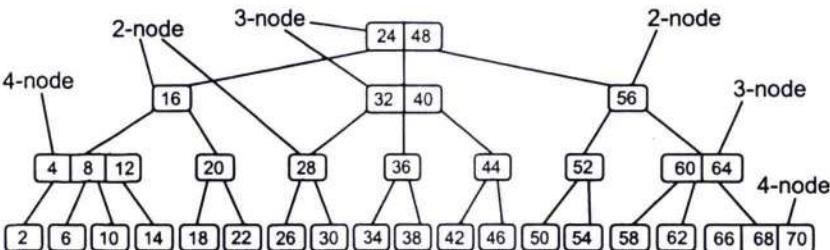
- Một nút chứa một khóa luôn luôn có đúng 2 nút con. Nó được gọi là 2-nút. (2-node)

- Một nút chứa hai khóa luôn luôn có đúng 3 nút con. Nó được gọi là 3-nút.

- Một nút chứa ba mục khóa luôn luôn có đúng 4 nút con. Nó được gọi là 4-nút.

- Nói một cách khác, một nút có n con tức là có n liên kết gọi là n -nút.

Như vậy, một nút không phải là luôn luôn có số nút con nhiều hơn 1 so với số khóa của nó. Nói cách khác, đối với mọi nút trong có số con là c và số khóa là k , thì: $c = k + 1$.



Hình 4.1. Tổ chức của cây 2-3-4

b. Các khóa nằm trong các nút khác nhau

Một đặc tính quan trọng của cây tim kiếm nhị phân là giá trị mọi khóa của cây con bên trái đều nhỏ hơn giá trị khóa của nút đang xét và tất cả các khóa của mọi nút ở cây con bên phải đều có giá trị của khóa lớn hơn hoặc bằng giá trị khóa ở nút đang xét.

Trong cây 2-3-4 tồn tại các tính chất sau:

1. Tất cả các nút con ở cây con thứ 1 của nút cha có các khóa nhỏ hơn khóa thứ nhất của nút cha.
2. Tất cả các nút con ở cây con thứ 2 của nút cha có các khóa lớn hơn khóa thứ nhất và nhỏ hơn khóa thứ hai của nút cha (nếu nút cha có khóa thứ hai).
3. Tất cả các nút con ở cây con thứ 3 (nếu có) của nút cha có các khóa lớn hơn khóa thứ hai và nhỏ hơn khóa thứ ba của nút cha (nếu nút cha có khóa thứ ba).
4. Tất cả các nút con ở cây con thứ 4 (nếu có) của nút cha có các khóa lớn hơn khóa thứ ba của nút cha.

Trong cây 2-3-4, tất cả các lá đều nằm trên cùng một mức. Các nút ở mức trên thường không đầy đủ, nghĩa là chúng có thể chứa chỉ 1 hoặc 2 khóa thay vì 3 khóa.

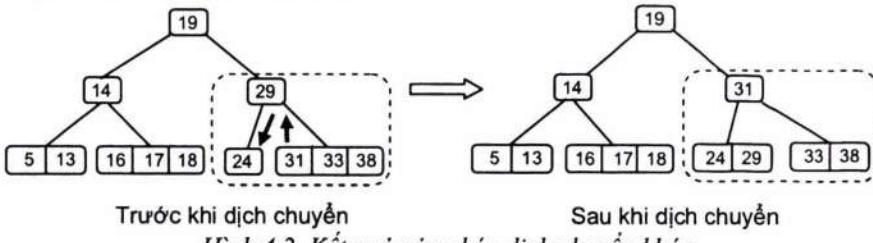
Lưu ý: cây 2-3-4 là cây cân bằng. Nó vẫn giữ được sự cân bằng ngay cả khi ta chèn thêm các phần tử theo thứ tự tăng dần hoặc giảm dần.

c. Khóa tiền bối (predecessor) và khóa hậu bối (successor)

Cũng như trong cây tìm kiếm nhị phân, trong cây 2-3-4 khóa tiền bối của một khóa k là khóa lớn nhất trong các khóa nhỏ hơn k, khóa hậu bối là khóa nhỏ nhất trong các khóa lớn hơn k. Theo cấu trúc của cây 2-3-4, để tìm khóa tiền bối và khóa hậu bối của khóa k trước hết tìm nút u chứa khóa k.

Nếu u là nút trong, giả sử k là khóa thứ m của u, khi đó khóa tiền bối là khóa cuối cùng của nút cực phải trong cây thứ m của nút u, còn khóa hậu bối là khóa đầu tiên trong nút cực trái của cây con thứ m+1 của nút u. Nếu khóa k nằm trong nút lá u, việc tìm khóa tiền bối và hậu bối có khó khăn hơn, tuy nhiên, trong các ứng dụng của cây 2-3-4 không dùng đến trường hợp này.

Trong hình 4.1 trên đây, khóa tiền bối của khóa 24 là khóa 22, còn khóa hậu bối của khóa 24 là khóa 26.



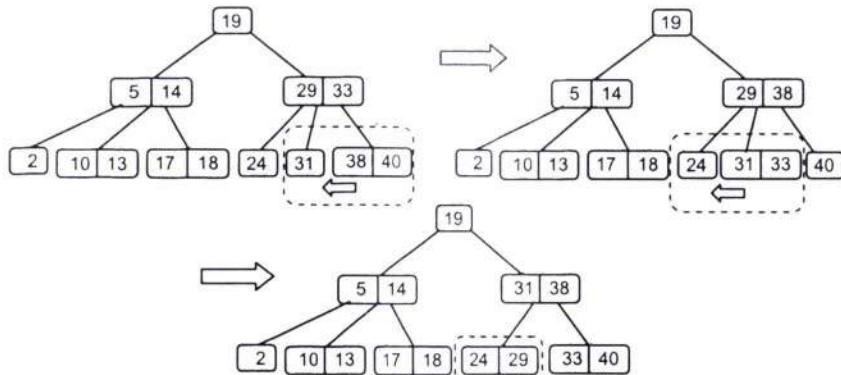
Hình 4.2. Kết quả của phép dịch chuyển khóa

4.1.3. Các phép biến đổi không làm thay đổi tính chất của cây 2-3-4

a. Dịch chuyển khóa

Phép biến đổi này chuyển một khóa từ một 3-nút hoặc 4-nút sang một nút anh em kè nó có ít hơn 3 khóa. Giả sử A là con thứ k của nút cha và A có 1 khóa là a1, B là nút thứ k+1 của nút cha và B có 3 khóa b1 < b2 < b3. Khóa thứ k của nút cha là ck. Khi đó ta có a1 < ck < b1 < b2 < b3. Phép dịch chuyển sẽ chuyển khóa ck từ nút cha xuống nút A và chuyển khóa đầu tiên b1 của nút B lên thay cho ck. Như vậy trong phép dịch chuyển này, số khóa của nút A tăng thêm một và số khóa của nút B giảm một. Điều kiện để thực hiện phép chuyển khóa là số khóa của nút A nhỏ hơn 3 và số khóa của nút B lớn hơn 1.

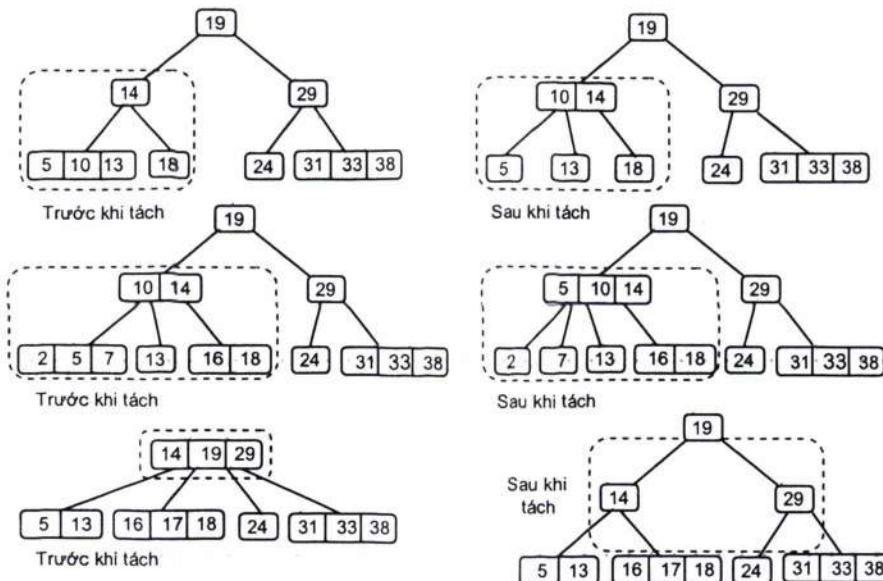
Nếu nút anh em liền kè với nút u chỉ có một nút có thể tăng số khóa của u từ một nút anh em không liền kè có nhiều hơn một khóa bằng 2 hoặc 3 phép dịch chuyển liên tiếp.



Để tăng số khóa của nút chứa 24 từ nút chứa 38, 40, trước hết dịch chuyển để tăng khóa của nút chứa 31 được nút chứa 31, 33. Sau đó dịch chuyển một khóa từ nút này sang cho nút chứa 24.

Hình 4.3. Minh họa quá trình thực hiện phép dịch chuyển khóa

b. Tách một nút



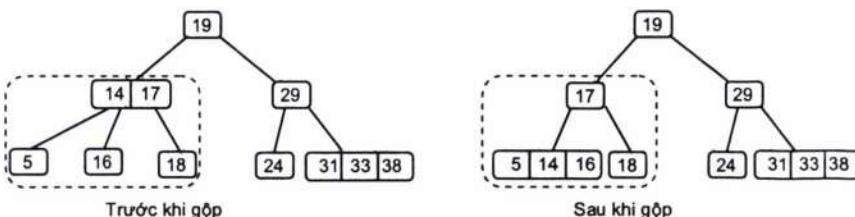
Tách nút gốc: Nút gốc không có cha nên phải thêm hai nút chứa khóa thứ nhất và khóa thứ ba, khóa thứ hai giữ lại

Hình 4.4. Quá trình tách nút

- Phép biến đổi này tách một nút thành hai nút, nghĩa là thêm một nút anh em với nó. Như vậy số con của nút cha tăng thêm một, do đó số khóa của nút cha cũng tăng lên một. Khóa tăng thêm này được lấy từ chính nút con sẽ được tách ra. Do đó để có thể tách một nút, nút đó phải có đúng 3 khóa $k_1 < k_2 < k_3$ (nó là 4-nút). Khi tách, chi khóa k_1 được giữ lại nút ban đầu (nút này trở thành 2-nút), k_3 được gán cho nút mới tạo thêm (là 3-nút), còn k_2 được thêm vào nút cha. Như vậy nút ban đầu và nút mới tạo ra là anh em liền kề. Nếu nút cha đã là 4-nút thì trước khi thêm k_2 vào nó cần tách chính nút cha này.

- Trường hợp nút cần tách là nút gốc. Vì nút gốc không có cha nên ngoài việc tách ra còn phải thêm một nút mới làm cha của nút ban đầu và nút mới tách ra.

c. Gộp hai nút



Hình 4.5. Quá trình gộp (trộn) nút

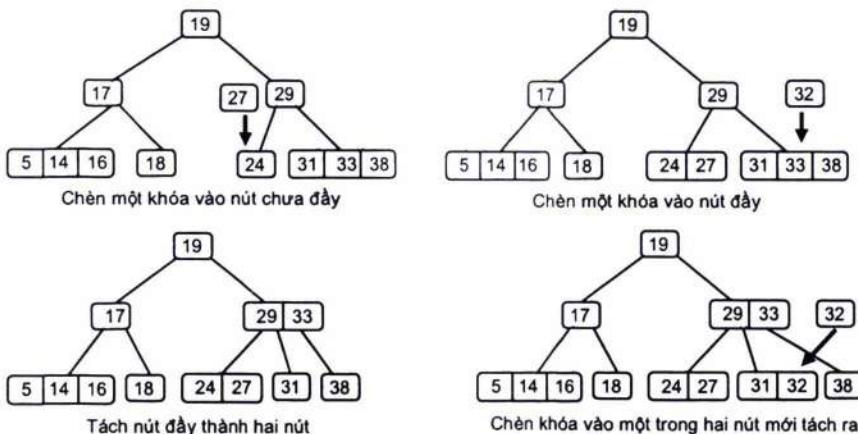
Phép gộp hai nút anh em liền kề thành một nút làm cho số con của nút cha giảm đi một, do đó cả số khóa của nút cha cũng giảm một. Khóa này được đưa cả vào nút mới gộp. Do đó chỉ có thể gộp hai anh em liền kề thành một nút khi cả hai là các 2-nút, nghĩa là mỗi nút chỉ có đúng một khóa, đồng thời cha của chúng phải có nhiều hơn một khóa. Sau khi gộp, nút ban đầu trở thành 4-nút, còn nút anh em được giải phóng.

d. Tìm một khóa trên cây 2-3-4

Để tìm một khóa k trên cây 2-3-4, trước hết ta tìm nó trong dãy khóa của nút gốc. Tại mỗi nút, nếu tìm thấy một khóa của nút bằng k thì trả về true và dừng quá trình tìm kiếm. Nếu không tìm thấy và nút đó là lá thì trả về false, còn nếu nút đó là nút trong và k nằm giữa khóa thứ m và $m+1$ thì tiếp tục tìm kiếm trong con thứ k của nút đó.

e. Chèn một khóa mới vào cây

Xem sơ đồ hình 4.6 (gồm 2 sơ đồ dưới đây)!



Hình 4.6. Quá trình chèn khóa mới

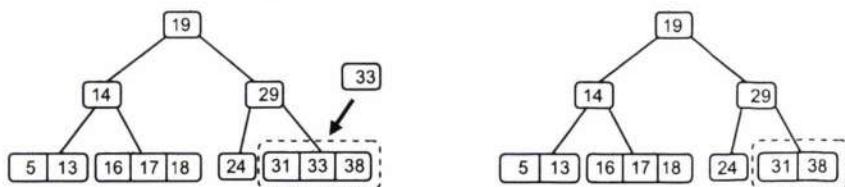
Để chèn một khóa vào một cây 2-3-4, trước hết tìm giá trị đó trong cây, nếu không có khóa này trong cây thì chèn nó vào nút lá gấp tại cuối quá trình tìm kiếm. Nếu nút này có ít hơn 3 khóa thì việc thêm khóa ấy vào nút này đơn giản chỉ là việc sắp xếp nó cùng với các khóa đã có theo thứ tự tăng. Nếu nút lá là 4-nút thì trước khi chèn ta tách nút đó ra. Điều phức tạp xảy ra khi nút cha của nút định tách cũng là 4-nút. Khi đó trước khi tách nút này phải tách nút cha của nó.

f. Xóa một khóa khỏi cây 2-3-4

Nếu phép chèn một khóa vào một nút phải giải quyết trường hợp tràn với nút đầy dẫn tới thao tác tách, nghĩa là thêm một nút, thì phép xóa phải giải quyết trường hợp cạn đối với 2-nút, khi đó việc giải phóng khóa ấy dẫn tới một nút rỗng, nghĩa là phải giải phóng nút này.

Phép xóa một khóa k khỏi cây 2-3-4 đòi hỏi những phân tích phức tạp hơn. Trước hết tìm nút chứa nó. Các trường hợp sau có thể xảy ra:

Trường hợp 1: Xem hình 4.7

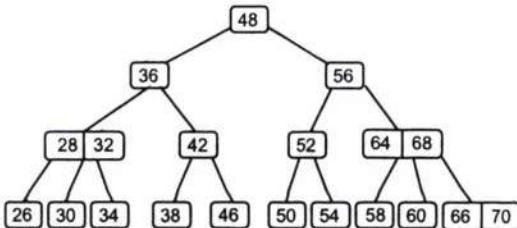
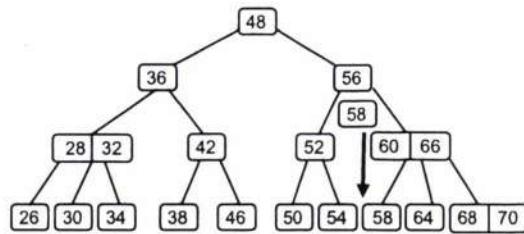


Case 1: Xóa khóa 33 trong nút lá không là 2-nút

Hình 4.7. Xóa khóa trường hợp 1

Khóa k (33) nằm trong nút lá u và u có nhiều hơn một khóa: giải phóng khóa k khỏi u.

Trường hợp 2: Xem hình 4.8



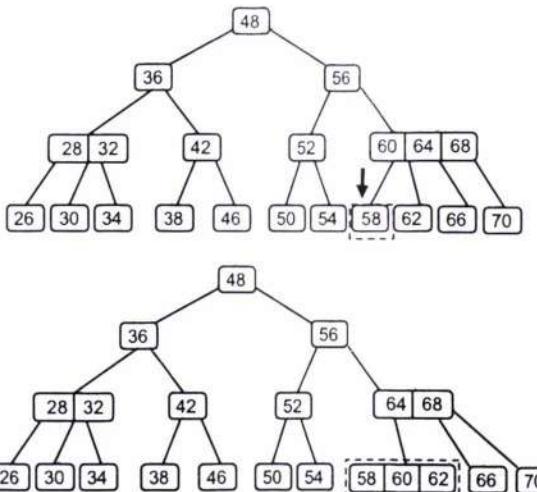
Case 2: Để xóa khóa 58 khỏi 2-nút nếu có nút anh em chứa nhiều hơn một khóa (nút chứa 68, 70) thì dịch chuyển dần để nút chứa 58 trở thành 3-nút.

Sau đó sẽ xóa khóa 58 tại nút này.

Hình 4.8. Xóa khóa trường hợp 2

Khóa k cần xóa (58) nằm trong nút lá u, u chỉ có một khóa và tồn tại nút anh em v của u có nhiều hơn một khóa thì bằng phép dịch chuyển dần có thể dịch chuyển một khóa của v đến u khiến u trở thành 3-nút và quay về trường hợp 1.

Trường hợp 3: Xem hình 4.9



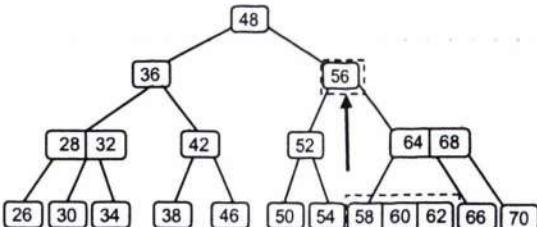
Case 3: Đề xóa nút 58 khỏi 2-nút nhưng tất cả các nút anh em đều là 2-nút: trước khi xóa đến nút anh em kề nó (nút chứa khóa 62) để có một 4-nút.

Sau đó giải phóng 58 trong 4 nút này.

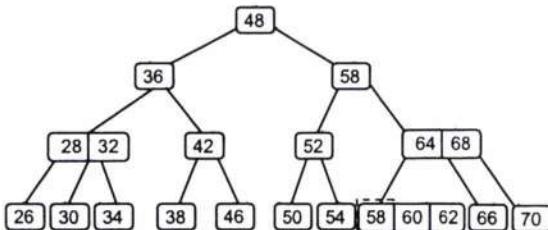
Hình 4.9. Xóa khóa trường hợp 3

Khóa k (58) nằm trong nút lá u, u chỉ có một khóa và tất cả các nút anh em của u chỉ có một khóa thì bằng phép gộp u với nút anh em kè nó sẽ khiến u trở thành 3-nút và quay về trường hợp 1.

Trường hợp 4, xem hình 4.10



Case 4: Để xóa khóa 56, tìm khóa tiền nhiệm là 54, khóa kế vị là 58. Vì khóa tiền nhiệm 54 nằm trong 2-nút nên chọn khóa kế vị 58 để thay cho khóa 56, sau đó xóa khóa 58 khỏi nút lá chứa nó.



Hình 4.10. Xóa khóa trường hợp 4

Khóa k (58) nằm trong nút u: Khi đó tìm khóa tiền bối hoặc khóa hậu bối của k (khóa này luôn nằm trong nút lá). Thay k bởi khóa đó và giải phóng khóa đó khỏi nút chứa nó (quay về trường hợp 1). Tuy việc dùng khóa tiền bối hay hậu bối đều được, nhưng nên chọn khóa nào trong chúng nằm tại nút có hai khóa trở lên, nếu cả hai đều nằm trong các 2-nút thì chọn khóa nào cũng được.

4.1.4. Mã giả mô tả một số phép cơ bản trên cây 2-3-4

Trong phần này ta sử dụng một số ký hiệu sau:

1. Nếu u là một nút thì u.keys biểu thị số khóa lưu trong u, khi đó số con của nó là u.keys+1, giá trị của các khóa được chứa trong mảng u.key[1..3];
2. Các nút con của nút u được trả đến bởi mảng con trỏ u.child[1..4];
3. Cha của nút u được trả bởi biến con trỏ u.parent, nếu u là nút gốc thì u.parent = Null, nếu u là con thứ m của nút cha thì m được lưu trữ bởi biến u.order=m;
4. Biến u.leaf = true/false tùy theo u là lá hay không phải lá.

a. Mã giả của phép dịch chuyển

Function Trans_keys(u,bleft)

//chuyển một khóa từ nút v là anh (nếu bleft = true)/em (nếu bleft=false) cho u nếu v có nhiều hơn một khóa và u có ít hơn ba khóa

```

if (u.keys>2) then return false;
if (bleft and u.order=1) or (!bleft and
u.order=u.parent.keys+1) then return false;
m=u.order;
if (bleft) then v=u.parent.child(m-1)
else v=u.parent.child(m+1);
  
```

```

if (v.keys < 2) then return false;
u.keys:=u.keys+1
if (bleft)      //Chuyển từ nút anh kê bên trái sang
begin
    for i:=2 to u.keys+1 do u.key(i)=u.key(i-1);
    u.key(1)=u.parent.key(m);
    u.parent.key(m)=v.key(v.keys);
end;
else           //Chuyển từ nút em kê bên phải sang
begin
    u.key(u.keys)=u.parent.key(m+1);
    u.parent.key(m+1)=v.key(1);
    for i:=1 to u.keys do
        v.key(i)=v.key(i+1);
end;
v.keys:=v.keys-1;
return true;

```

b. Mã giả của phép tách nút

```

Function SplitNode(u);
if (u.keys<3) return false;
if (u.parent == Null)
//tách nút gốc
    (NewNode(v1); v1.key(1):=u.key(1); v1.keys:=1;
    v1.parent:=u;
    NewNode(v2); v2.key(1):=u.key(3); v2.keys:=1;
    v2.parent:=u;
    u.child(1).parent:=v1; u.child(2).parent:=v1;
    v1.child(1)=u.child(1);
    v1.child(2)=u.child(2);
    u.child(3).parent:=v2; u.child(4).parent:=v2;
    v2.child(1)=u.child(3);
    v2.child(1)=u.child(1);
    u.key(1)=u.key(2); u.keys:=1;

```

```

    }
else      //tách nút không là gốc
{
    if (u.parent.keys>2)  SplitNode(u.parent);
//gọi đệ quy
    m= u.order;
    u.parent.keys=u.parent.keys+1;
    for i:=u.parent.keys-1 downto m do
    u.parent.key(i+1)=u.parent.key(i);
    for i:=u.parent.keys downto m+1 do
    u.parent.child(i+1):= u.parent.child(i);
    u.parent.key(1)=u.key(2);
    NewNode(v); v.keys:=1; v.key(1):= u.key(3);
    v.parent=u.parent; u.parent.child(2):=v;
    v.child(1):=u.Child(3); v.child(2):=u.Child(4);
    u.keys:= 1;
}

```

c. Mã giả của phép gộp nút

```

Function Fusion(u);
if u.keys>1 then return false;
    if u.parent = Null then return false;
    m:=u.order;
    if m = 1 then v:= u.parent.child(2) else v:=
u.parent.child(m-1);
    if v.keys>1 then return false;
    u.keys:=3;
    if u.parent= Null then return false;
    if u.parent.keys=1 then TwoNodeAddKey(u.parent);
    if m:=1 then begin
        u.key(2)=u.parent.key(1); u.key(3):=v.key(1);
        u.child(3):=v.child(1);
        u.child(4):=v.child(2); v.Child(3).parent:=u;
        v.Child(4).parent:=u;
        for i:=1 to u.parent.keys-1 do u.key(i):=
u.key(i+1);
    end;
end;

```

```

        for i:=2 to u.parent.keys do
            u.parent.child(i):=u.parent.child(i+1)
    end;
    else begin
        u.key(3):= u.key(1); u.key(1):=v.key(1);
        u.key(2):= u.parent.key(m-1);
        u.child(3):=uChild(1);u.Child(4):=u.Child(2);
        u.child(1):=v.child(1);
        u.child(2):=v.child(2); v.child(1).parent:=u;
        v.child(2).parent:=u;
        for i:=m to u.parent.keys-1 do u.key(i):=
            u.key(i+1);
        for i:=m to u.parent.keys do
            u.parent.child(i):=u.parent.child(i+1);
            u.parent.child(m-1):=u;
    end;
u.parent.keys:=u.parent.keys-1;
dispos(v);

```

d. Mã giả tìm nút anh em có nhiều hơn một khóa

```

Function FindSubling(u);
if u.parent= Null then return 0;
m:=u.order;
k:=m+1
while (k≤ u.parent.keys+1) and (u.parent.child(k).keys<2)
do k:=k+1;
if k ≤ u.parent.keys+1 then return k;
else while (k≥1) and (u.parent.child(k).keys<2) do
k:=k-1;
return k;

```

Nếu hàm trả về 0 thì tất cả các nút anh em đều là 2-nút

e. Mã giả tăng số khóa cho 2-nút

Khi xóa một khóa trong 2-nút, trước hết phải dùng phép chuyển khóa hoặc phép dồn nút để tăng số khóa của nút này. Nếu tìm được một khóa anh em có nhiều hơn một khóa có thể dùng phép dịch chuyển, nếu không thì dùng một phép gộp nút.

```

Function TwoNodeAddKey(u);
If u.keys>1 then return false ;
if u.parent=null then fusion(u);
else begin
  k= FindSubling(u);
  if k=0 then
    fusion(u);
  else
    if k< u.order then
      for i:= k+1 to u.order do
        TransKey(u.parent.Child(i),true);
    else
      for i:= k-1 downto u.order do
        TransKey(u.parent.Child(i),false);
end;

```

f. Mã giả tìm kiếm một khóa trong cây

Tìm một khóa trong một nút

Giả sử u có n khóa. Đặt thêm các phần tử khóa hai đầu $u.key(0) = \infty$ và $u.key(n+1) = \infty$. Khi tìm khóa k trong nút u có thể hoặc $k = u.key(j)$ với $1 \leq j \leq n$ hoặc k thuộc một trong $u.keys+1$ khoảng $(u.key(j), u.key(j+1))$, $j = 0, \dots, n$. Trong trường hợp thứ nhất hàm trả về chỉ số khóa tìm thấy là j , trong trường hợp sau hàm trả về chỉ số của cận sau trong khoảng nó tìm thấy với dấu âm.

```

Function NodeSearchKey(u, k);
for j:=1 to u.keys do if k=u.key(j) return j;
j:=1;
while k > u.key(j) and j < n do j:= j+1;
return j+1;
//Tim một khóa trong cây
Function TreeSearchKey(k; var v);
v:= Root.
k:=NodeSearchKey(v, k);
while k<0 and !u.leaf do begin
  v:= v.child(abs(k));
  k:=NodeSearchKey(v, k); end;
if k>0 return true else return false;

```

```

//Tìm tiền bối và hậu bối của một khóa
Nút chứa khóa tiền bối của khóa thứ m trong nút u
Function Predecessor(u,m);
v:= u.Child(m);
while !v.leaf do v:=v.child(v.keys-1);
return v; {khóa cuối của v là khóa cần tìm}.
Nút chứa khóa hậu bối của khóa thứ m trong nút u
Function Successor(u,m)
v=u.child(m+1);
while !v.leaf do v:=v.child(1);
return v; {khóa thứ nhất của v là khóa cần tìm}.

```

g. Mã giả chèn một khóa vào cây 2-3-4

Mã giả chèn khóa k vào cây 2-3-4 sử dụng thủ tục đệ quy với nút gốc.

```

Function NodeInsert(u,k);
if u= Null then begin
    NewNode(v);
    v.Keys=1; v.key(1) := k;
    root:= v;
    v.parent := Null;
end;
else begin
    m=NodeSearchKey(u, k)
    if u.leaf then
        if u.keys<3 then begin
            if m>0 then return 0
        else begin
            m=-m; u.keys:=u.keys+1;
            for j:=u.keys downto m+1 do u.key(j):= u.key(j-1);
            u.key(m) := k;
        end;
    end;
    else begin {u có 3 khóa}
        m:=NodeSearchKey(u,k);
        SplitNode(u);
    end;
end;

```

```

v:= u.parent.child(u.order+1);
    case of m
    1: u.keys:=2; u.key(2):= u.key(1); u.key(1):=k;
    2: u.keys:=2; u.key(2):= k;
    3: v.keys:=2; v.key(2):= v.key(1); v.key(1):=k;
    4: v.keys:=2; v.key(2):= k;
end;
else begin { u không là lá }
    if m>0 then return false
    else NodeInsert(u.child(m), k)
end;
end;

```

h. Mã giả xóa khóa khỏi cây 2-3-4

```

//Xóa một khóa ở nút lá
Function LeafDelete(u, k);
if !u.leaf then return false;
m:=NodeSearchKey(u, k)
if m<0 then return false;
for i:= m to u.keys-1 do u.key(i):=u.key(i+1)
u.keys:= u.keys-1;

//Xóa một khóa trong cây gốc là nút u
Function NodeDelete(u, k);
m=NodeSearch(u, k);
if m>0 then begin {tìm thấy k trong nút u}
    if u.left then begin {nếu u là lá}
        if (u.keys=1) and u=root then begin
            dispos(u); return true; end
        else begin
            if u.keys=1 then fusion(u);
            leafdelete(u, k);
        end;
    end {nếu u là lá}
    else begin {tìm thấy k trong nút u là nút trong}

```

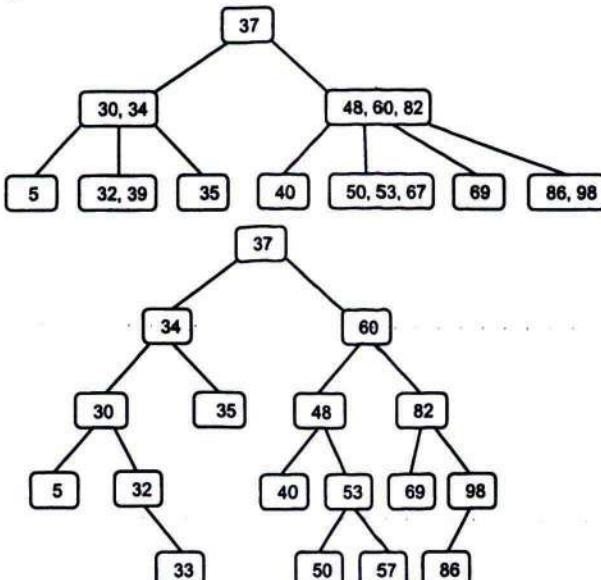
```

v:= predecessor(u,m);
if v.keys=1 then v:=successor(u,m);
if v.keys=1 then fusion(u);
leafdelete(v,k);
end; {tim thấy k trong nút u là nút trống}
end { tim thấy k trong nút u }
else begin {m< 0 không tìm thấy k trong nút u}
m:=-m;
v:= u.child(m);
if v # Null then NodeDelete(v, k)
else return false; {không có khóa k trong cây gốc u}
end;
end; {m< 0: không tìm thấy k trong nút u}

```

4.1.5. Đánh giá tính hiệu quả của cây 2-3-4

Với cùng số lượng mục dữ liệu thì cây 2-3-4 có số mức ít hơn (tức là chiều cao nhỏ hơn) so với cây đòn đồ đen (hình 4.11). Điều này khiến cho thời gian tìm kiếm trên cây 2-3-4 nhanh hơn cây đòn đồ đen.



Hình 4.11. Cây 2-3-4 và cây đòn đồ đen tương ứng

Ví dụ: ta xét cây 2-3-4 với nút có tối đa 4 con và giả sử các nút đều đầy (lưu đủ mục dữ liệu). Khi đó chiều cao của cây này tỷ lệ với $\log_4 N$ (N số nút của cây). Trong khi đó chiều cao của cây đòn tay tỷ lệ với $\log_2 N$. Như vậy chiều cao của cây 2-3-4 nhỏ hơn một nửa so với chiều cao cây đòn tay.

Tuy nhiên có nhiều mục dữ liệu trong một nút thì nếu tìm kiếm theo thuật toán tuyến tính sẽ làm tăng thời gian tìm kiếm. Ta gọi M là số mục dữ liệu trung bình trên một nút trong cây 2-3-4 thì thời gian tìm kiếm trên cây 2-3-4 sẽ là $M \cdot \log_4 N$.

Vấn đề hiệu quả còn được xét trên phương diện lưu trữ các mục dữ liệu của cây 2-3-4. Ở tình huống tối đa, mỗi nút trong cây 2-3-4 cần lưu trữ 3 tham chiếu đến các mục dữ liệu và 4 tham chiếu đến các con của nó. Giả sử ta lưu trữ các tham chiếu bằng mảng ứng với một vùng nhớ xác định được cấp phát. Sau đây ta sẽ thấy không phải tất cả vùng nhớ này sẽ được dùng hết. Đề cử thể ta xét trường hợp một nút có 2 mục dữ liệu sẽ bỏ trống (không dùng) 1/3 vùng nhớ để lưu dữ liệu và bỏ trống 1/4 vùng nhớ lưu tham chiếu của các nút con.

Ta thử khắc phục sự lãng phí này bằng cách dùng danh sách liên kết đơn để cài đặt cây 2-3-4. Song tổng chi phí cho một DSKĐ so với mảng có thể làm tăng hiệu quả không đáng kể.

Trong khi đó cây đòn tay chỉ có vài nút có một con vì thế hầu hết vùng nhớ lưu các tham chiếu của các nút con đều được sử dụng. Thêm nữa mỗi nút chỉ có một mục dữ liệu, điều này làm cho cây đòn tay sử dụng không gian nhớ hiệu quả hơn cây 2-3-4.

Bù lại hai nhược điểm trên đây, cài đặt cây 2-3-4 trên các ngôn ngữ Java hay C++ đơn giản hơn cây đòn tay.

4.2. CÂY ĐA NHÁNH TÌM KIẾM

4.2.1. Định nghĩa cây đa nhánh tìm kiếm

Cây đa nhánh tìm kiếm (Multiway Search Tree-MST) bậc d là cây mà mỗi nút có tối đa d cây con và các mục dữ liệu được sắp theo trật tự tăng (xem chi tiết mục sau).

4.2.2. Các tính chất của MST

a. Tính chất 1

Gọi số các cây con của một nút là numsubtr ($\text{numsubtr} \leq d$) và số khóa của nút ấy là numk thì trong MST luôn nghiệm đúng quan hệ sau:

$$\text{numsubtr} = \text{numk} + 1 \Leftrightarrow \text{numk} = \text{numsubtr} - 1$$

b. Tính chất 2

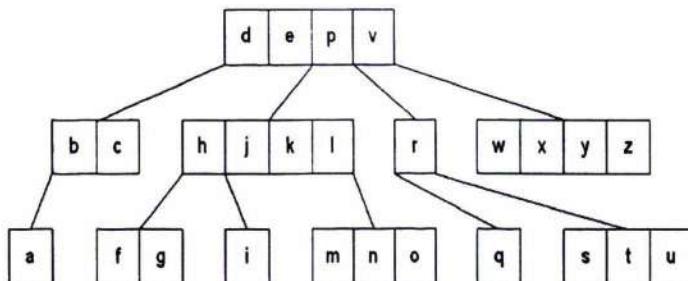
Tất cả các nút thuộc cây con có nút gốc là nút con thứ i thì mọi giá trị khóa (mục dữ liệu) của chúng lớn hơn giá trị khóa $k[i-1]$ và nhỏ hơn giá trị khóa $k[i]$.

c. Tính chất 3

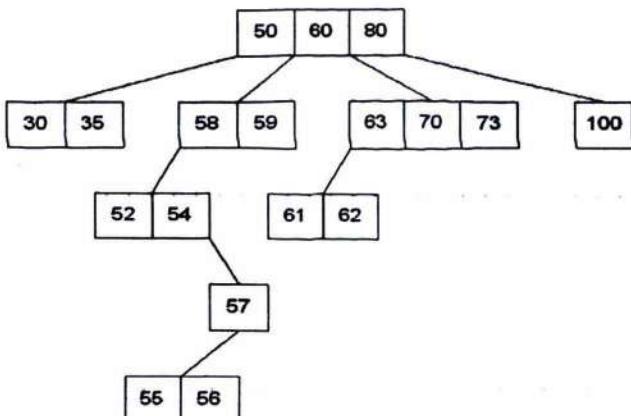
Tất cả các nút thuộc cây con có nút gốc là nút con thứ numsubtr thì có giá trị khóa lớn hơn giá trị khóa thứ $\text{numsubtr}-1$.

Chú thích: Các giá trị khóa (các mục dữ liệu) trong cây MS thường được lưu trữ bởi mảng. Ở đây ta dùng mảng $k[i]$.

Các hình 4.12 và 4.13 dưới đây là hai sơ đồ minh họa cây MS.



Hình 4.12. Cây MS bậc 5



Hình 4.13. Cây MS bậc 4

4.3. B-CÂY

4.3.1. Nguồn gốc chữ “B” trong tên gọi B-cây (B-Tree)

Rudolf Bayer và Ed McCreight đã tìm ra B-Tree trong thời gian làm việc tại Phòng Nghiên cứu Khoa học của hãng Boeing (Boeing Scientific Research Laboratories) vào năm 1971. Song chưa bao giờ hai cha đẻ này của B-Tree giải thích một điều gì về chữ B dùng trong thuật ngữ trên.

Chuyên gia điện toán Douglas Comer cũng có nhận xét tương tự và giải thích thêm: “Gốc gác của “B-Tree” chưa bao giờ được hai tác giả giải thích. Nhưng như chúng ta thấy chữ B đều có thể lấy từ: “Balanced”, “Broad”, hoặc “Bushy”. Một đề xuất khác của D. Comer là “chữ “B” có thể lấy từ chữ cái đầu của Boeing. Ngoài ra vì những đóng góp của Bayer nên tôi nghĩ B-Tree là “Bayer-Tree” cũng xứng đáng”.

Còn Donald Knut (một chuyên gia nổi tiếng về thuật toán) trong lần đọc chuyên đề vào tháng 5/1980 về chủ đề “CS144C classroom lecture about disk storage and B-trees”, thì phỏng đoán “nước đôi”: “Chữ B có thể là chữ đầu của “Boeing” hoặc là chữ đầu của cái tên Bayer”.

4.3.2. Giới thiệu chung về B-cây

Trong khoa học máy tính, B-cây là một cấu trúc dữ liệu dạng cây cho phép tìm kiếm, truy cập tuần tự, chèn, xóa trong thời gian lôgarit. B-cây là một tổng quát hóa của cây nhị phân tìm kiếm, trong đó một nút có thể có nhiều hơn hai con. Không như cây nhị phân tìm kiếm và các cây khác mà ta đã tìm hiểu ở trên (chi lưu dữ liệu ở bộ nhớ trong - RAM), B-cây được tối ưu hóa cho các hệ thống đọc và ghi số lượng dữ liệu lớn hoặc rất lớn ở bộ nhớ ngoài. Nó thường được dùng trong các cơ sở dữ liệu và hệ thống tệp tin khi mà dung lượng dữ liệu cần lưu trữ vượt nhiều không gian nhớ của RAM.

Trong B-cây, các nút trong (nút không lá) có thể có số lượng nút con khác nhau, giới hạn trong một khoảng nhất định. Khi dữ liệu được chèn vào hoặc xóa đi từ một nút, số nút con của nó thay đổi. Khi đó, để duy trì số nút con trong khoảng đã định, các nút trong có thể được hợp lại làm một hoặc tách đôi. Vì số lượng nút con có thể nằm trong một khoảng lớn, B-cây không cần tái cấu bằng thường xuyên như cây nhị phân tìm kiếm, nhưng lại sử dụng bộ nhớ lãng phí hơn do các nút không chứa tối đa dữ liệu.

B-cây duy trì tính cân bằng bằng cách đảm bảo các nút lá đều có cùng độ sâu. Chiều cao của cây tăng dần dần khi các nút mới được chèn vào cây, nhưng con số này thay đổi rất chậm. Khi chiều cao của cây tăng lên, độ sâu của tất cả các nút lá tăng lên cùng một lúc.

B-cây có lợi thế hơn các cấu trúc dữ liệu tìm kiếm khác khi thời gian truy cập lớn hơn nhiều lần thời gian đọc dữ liệu liên tiếp nhau. Điều này thường xảy ra khi các nút được lưu trên bộ nhớ ngoài. Bằng cách tăng số lượng nút con của mỗi nút, chiều cao của cây giảm xuống và số lần truy cập cũng giảm. Thêm vào đó, số thao tác tái cân bằng cây cũng giảm đi. Thông thường, tham số d (bậc của cây quyết định số khóa của mỗi nút) được chọn tùy theo lượng thông tin trong mỗi khóa và kích thước mỗi khối đĩa.

4.3.3. Định nghĩa B-cây

Có nhiều định nghĩa khác nhau của B-cây trong các tài liệu. Sau đây là định nghĩa đủ ngắn gọn và rõ ràng của Knut:

B-cây là cây đa nhánh tìm kiếm bậc d (degree) có thêm các tính chất sau:

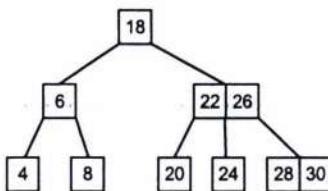
1. Mọi nút lá đều có cùng một độ sâu (tức là ở cùng một mức).
2. Mỗi khóa tại nút x có giá trị nằm giữa giá trị các khóa tại hai cây con tương ứng của x : khóa thứ i của nút x lớn hơn hoặc bằng mọi khóa ở cây con thứ i của x và nhỏ hơn hoặc bằng mọi khóa ở cây con thứ $i+1$ của x :

$$k(\text{subtree}_i, x) \leq k(i, x) \leq k(\text{subtree}_{i+1}, x)$$

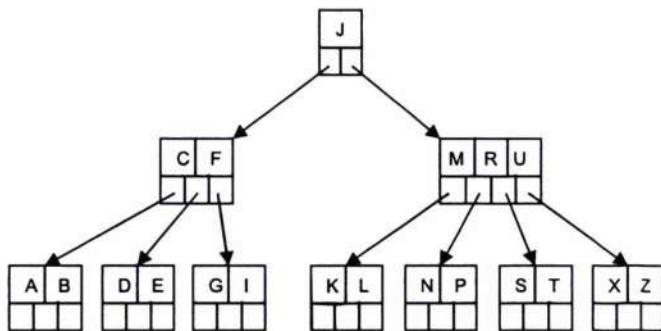
3. Mọi nút trong (trừ gốc) có tối đa d cây con và có ít nhất $d/2$ cây con khác rỗng.

4. Nút gốc có nhiều nhất d con, ít nhất 2 con không phải là lá hoặc không có con nào.

5. Một nút không phải lá có số khóa là numkey thì có số cây con là $\text{numkey}+1$ (Nói ngược lại: Một nút không phải lá có numsubtr cây con thì có số khóa là $\text{numsubrt}-1$).



Hình 4.13 B-cây bậc 3, 3 mức



Hình 4.14 B-cây bậc 5, 3 mức

4.3.4. Các biến thể (Variants) của B-cây

Thuật ngữ B-cây có thể được dùng để chỉ một thiết kế cụ thể, cũng có thể được dùng để chỉ một lớp các thiết kế. Theo nghĩa hẹp, B-cây lưu một số khóa ở các nút trong và không lưu bản sao của các khóa đó ở nút lá. Theo nghĩa rộng, nó còn bao gồm những biến thể khác như B+-cây hay B*-cây.

- Trong B+-cây, các nút trong lưu bản sao của khóa. Tất cả các khóa, cùng với dữ liệu đi kèm được lưu ở các nút lá. Ngoài ra các nút lá còn có con trỏ đến các nút lá kế bên để tăng tốc truy cập tuần tự.

- B*-cây thực hiện nhiều thao tác tái cân bằng hơn để lưu trữ dữ liệu dày đặc hơn.

- Mỗi nút trong khác gốc phải đầy tối hai phần ba thay vì chỉ một nửa như trong B-cây.

4.3.5. Mã mô tả các thao tác trên B-cây

a. Định nghĩa B-cây và các thao tác trên nó

```

class BTreeNode //Khai báo cấu trúc nút trên B-cây
{
    int *keys;      //Mảng chứa các khóa
    int d;          //Bậc của cây (định nghĩa số lượng khóa)
    BTreeNode **C; //Mảng chứa các con trỏ đến các nút con
    int n;          //Số hiệu hiện tại của khóa
  
```

```

bool leaf; //Kiểm tra một nút có phải là lá không
public:
BTreeNode(int _d, bool _leaf); //Hàm tạo cây
void traverse(); //Hàm duyệt cây
BTreeNode *search(int k); //Hàm tìm một khóa, hàm trả về NULL nếu khóa k không có trên cây
int findKey(int k); //Hàm tìm khóa ≥ k cho trước
void insertNonFull(int k); //Hàm chèn khóa mới vào nút chưa đầy
void splitChild(int i, BTreeNode *y); //Hàm này được gọi để tách nút khi nút đầy
void remove(int k); //Hàm xóa một khóa ở cây con có gốc là nút hiện hành
void removeFromLeaf(int idx); //Hàm xóa khóa ở vị trí idx trong lá
void removeFromNonLeaf(int idx); //Hàm xóa khóa ở vị trí idx thuộc nút không phải lá
int getPred(int idx); //Hàm xác định tiền bối của một khóa ở vị trí idx của nút
int getSucc(int idx); //Hàm xác định hậu bối của một khóa ở vị trí idx của nút
void fill(int idx); //Hàm lấp đầy nút con ở vị trí idx của mảng C[] nếu nút này có ít hơn d-1 khóa
void borrowFromPrev(int idx); //Hàm mượn một khóa ở nút trước C[idx-1] đặt nó vào nút C[idx]
void borrowFromNext(int idx); //Hàm mượn một khóa ở nút kế sau C[idx+1] đặt vào nút C[idx]
void merge(int idx); //Hàm trộn (gộp) con thứ idx của một nút với con thứ idx+1
friend class Btree; /*Hàm bạn để truy cập vào các thành viên riêng trong lớp các hàm của BTee */
};

b. Khởi tạo cây và các phương thức của nó
class BTee
{
    BTreeNode *root; //Con trỏ gốc
}

```

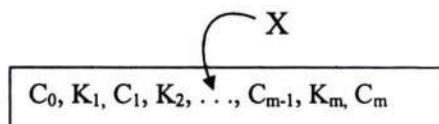
```

int d;                                // Bậc của cây
public:
BTree(int _d)                         //Hàm tạo (constructor),
                                         khởi trị cây rỗng
{
    root = NULL;
    d = _d;
}
void traverse() //Hàm duyệt cây nếu cây không rỗng
{
    if (root != NULL) root->traverse();
}
BTreeNode* search(int k)                //Hàm tìm khóa
{
    return (root == NULL)? NULL : root->search(k);
/* Nếu cây rỗng trả về NULL, ngược lại trả về khóa k nhờ
hàm search(k)*/
}
void insert(int k);                   //Hàm chèn khóa mới vào cây
void remove(int k);                  //Hàm xóa một khóa trên cây
};

```

c. Tìm kiếm trên B-cây

Mô tả thuật toán tìm bằng ngôn ngữ tự nhiên, xem sơ đồ:



Hình 4.15. Tìm kiếm giá trị X

Xét nút trong hình 4.15, khóa cần tìm là X. Giả sử dữ liệu của nút lưu ở bộ nhớ trong. Với bậc d đủ lớn ta sử dụng phương pháp tìm kiếm nhị phân, nếu d nhỏ ta sử dụng phương pháp tìm kiếm tuần tự.

Nếu X không tìm thấy sẽ có 3 trường hợp sau xảy ra:

- + $K_i < X < K_{i+1}$: Tiếp tục tìm kiếm trên cây con C_i
- + $K_m < X$: Tiếp tục tìm kiếm trên C_m
- + $X < K_1$: Tiếp tục tìm kiếm trên C_0

Quá trình này tiếp tục cho đến khi nút X được tìm thấy. Nếu đã đi đến nút lá mà vẫn không tìm thấy khóa, việc tìm kiếm là thất bại.

Code C++ mô tả thuật toán tìm kiếm

```
BTreeNode *BTreeNode::search(int k)
{
    //Trước hết tìm khóa ≥ k
    int i = 0;
    while (i < n && k > keys[i]) //Chừng nào mà chưa
        duyệt hết các nút trên cây và k còn lớn hơn keys[i] thì
        lặp lại các lệnh sau:
    i++;
    if (keys[i] == k)           //Nếu giá trị khóa ở vị trí i
                                //bằng k
        return this;           //thì trả về nút đó
    if (leaf == true)          //Nếu không tìm thấy khóa cần
                                //tim và nút chứa khóa ấy là lá
        return NULL;           //thì trả về NULL, rồi đi tiếp
                                //đến con thích hợp để tìm khóa
    return C[i]->search(k);   //Trả về k ở vị trí do
                                //con trỏ C[i] trả tới
}
```

d. Chèn khóa mới vào B-cây

Mô tả thuật toán chèn vào B-cây bằng ngôn ngữ tự nhiên.

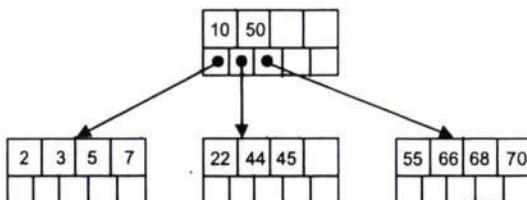
Trước khi mô tả thuật toán chèn bằng code, ta hãy xem cơ chế chèn khóa mới vào B-cây như thế nào và diễn đạt bằng ngôn ngữ tự nhiên. Để chèn một giá trị k vào B-cây có 3 bước:

- Dùng hàm tìm để tìm một lá có thể thêm k vào được.
- Thêm k vào nút này tại vị trí thích hợp.
- Nếu có ít hơn $d-1$ giá trị ở nút vừa thêm k, thì kết thúc công việc!

Nếu có d giá trị sau khi thêm k, thì nút này bị tràn. Để khắc phục “sự cố” này, ta tách nút ấy thành 3 phần:

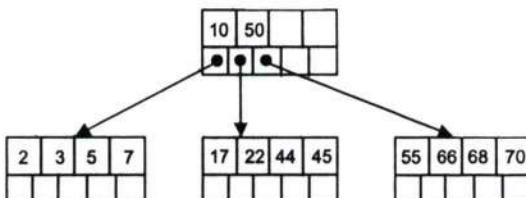
- Phần trái: chứa $(d-1)/2$ giá trị đầu tiên (d là bậc của B-cây).
- Phần giữa: chứa giá trị giữa (tức là giá trị tại vị trí $1+((d-1)/2)$).
- Phần phải: chứa $(M-1)/2$ giá trị cuối cùng.

Ví dụ: Đầu tiên ta cần chèn 17, sau đó lần lượt là 6, 21, 67 vào B-cây có bậc $d=5$, 2 mức dưới đây (hình 4.16)



Hình 4.16. B-cây bậc 5, 2 mức

Chèn 17: Thêm 17 vào lá giữa, không bị tràn ô. Kết thúc việc chèn (hình 4.17 a).

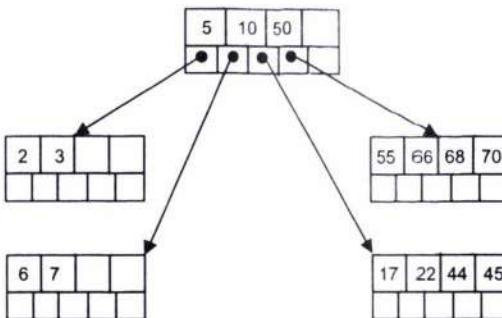


Hình 4.17 a). Chèn 17

Chèn 6: Ta sẽ thêm 6 vào vị trí ở bên trái nhất của lá (cận trái). Nút này bị tràn, vì vậy ta phải tách nó thành 3 phần như trên đã nói:

- Phần trái có: [2 3]
- Phần giữa có: 5
- Phần phải có: [6 7]

Giờ ta phải đẩy giá trị 5 (ở phần giữa) lên vị trí cận trái của nút gốc. Rồi đặt 6 bên trái 7 (vị trí cận trái của phần phải vừa tách). Và khi ấy các nút 2, 3 là con của 5 (hình 4.17 b).



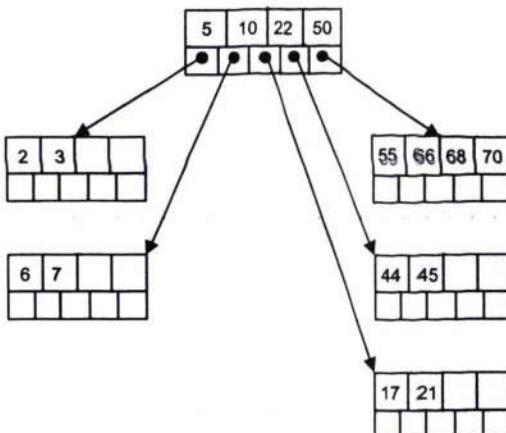
Hình 4.17 b). Chèn 6

Nút ở trên cùng (gốc) không bị tràn. Kết thúc việc.

Chèn 21: Thêm 21 vào lá giữa của B-cây đã cho (hình 4.16). Nút này bị tràn, nên ta tách nó thành 3 phần:

- Phần trái: [17 21]
- Phần giữa: 22
- Phần phải: [44 45]

Để có chỗ hợp lệ thêm 21 vào (bảo toàn các đặc tính của B-cây) ta phải đẩy 22 lên vị trí áp chót của nút gốc, rồi đặt 21 vào chỗ cũ của 22 (hình 4.17 c))



Hình 4.17 c). Chèn 21

Khi ấy nút gốc vẫn không bị tràn nên công việc kết thúc.

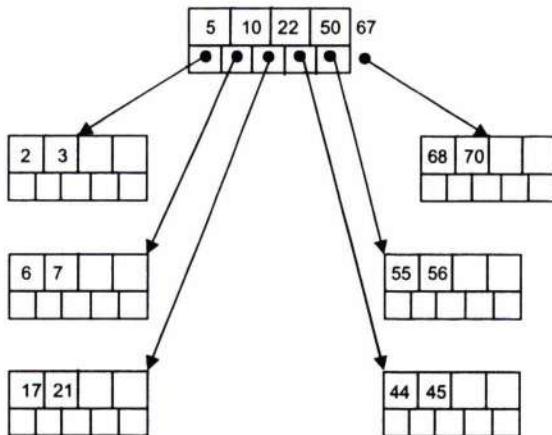
Chèn 67: Thêm giá trị này vào nút con phải của gốc thuộc cây ở hình 4.17 c) thì nút này bị tràn, nên ta cần tách nó như sau:

- Phần trái: [55 66]

- Phần giữa: 67

- Phần phải: [68 70]

Xem hình 4.17 d) dưới đây.



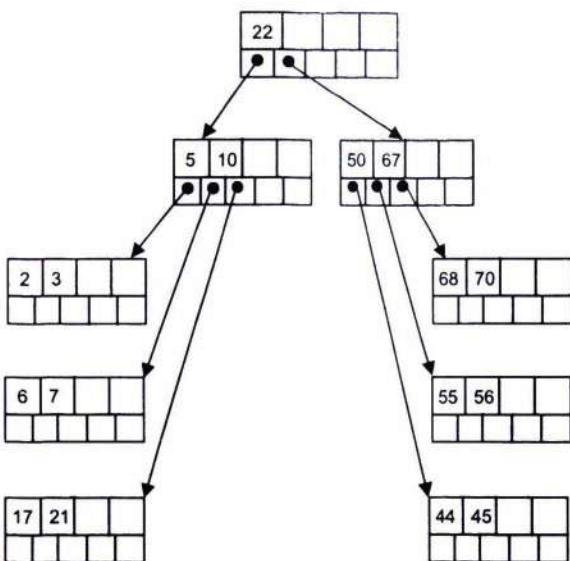
Hình 4.17 d). Tách nút con phải của gốc

Tất nhiên phần trái và phải giờ trở thành các nút. Thoạt nhìn, về mặt giá trị ta thấy vị trí cận phải của nút cha của nút [68 70] là vị trí thích hợp để nạp 67 vào. Tuy nhiên nút cha này (chính là nút gốc) đã đầy. Nếu thêm 67 vào thì nó tràn, vì vậy ta cần tách nó thành:

- Phần trái: [5 10]

- Phần giữa: 22

- Phần phải: [50 67]



Hình 4.17 e). Tách gốc và chèn 67

Sau lần tách này, các nút [5 10] và [50 ...] thành con trái và phải của gốc. Bây giờ ta đặt 67 vào vị trí bên phải giá trị 50. Các cấu hình còn lại giữ nguyên như hình 4.17 d). Đến đây thuật toán chèn các số đã cho vào B-cây kết thúc. Kết quả được cây như ở hình 4.17 e).

e. Code C++ mô tả hàm chèn chính để chèn khóa vào B-cây

```
void BTTree::insert(int k)
{
    if (root == NULL)           // Nếu cây rỗng (thì)
    {
        root = new BTTreeNode(t, true); // Cấp phát vùng
                                         // nhớ cho gốc
        root->keys[0] = k;          // (rồi) chèn khóa k
                                     // vào nút gốc
        root->n = 1;              // Cập nhật số khóa tại gốc
    }
    else                        // Ngược lại nếu cây không rỗng (thì)
    {

```

```

if (root->n == 2*d-1)           //Nếu gốc đầy, thì tăng
                                chiều cao lên
{
    BTreeNode *s = new BTreeNode(d, false) // (Bằng
                                         cách) cấp phát vùng nhớ cho nút mới
    s->C[0] = root;      //Đổi gốc cũ thành con của
                          gốc mới
    s->splitChild(0, root);     //Tách gốc cũ và
                                chuyển một khóa đến gốc mới
// (Xem hàm tách ở phần cài đặt!)
//Bây giờ gốc mới có 2 con và ta sẽ chèn khóa mới
    int i = 0;
    if (s->keys[0] < k) //Nếu khóa ở gốc nhỏ thua
                          khóa k
        i++;            //thì tăng chỉ số i lên 1 (rồi)
        s->C[i]->insertNonFull(k); //Gọi hàm chèn
                                      insertNonFull (xem dưới đây) để chèn k
        root = s;          //(rồi) đổi gốc là s
    }
    else
        root->insertNonFull(k); /*Ngược lại gốc không
đầy thì gọi insertNonFull để chèn vào gốc*/
}

```

f. Code C++ mô tả hàm chèn vào nút không đầy (Giả sử rằng khi gọi hàm này, nút cần chèn khóa mới không đầy)

```

void BTreeNode::insertNonFull(int k)
{ int i = n-1;           //Khởi trị cho chỉ số của nút bên
                           //phải nhất (tận cùng bên phải)
  if (leaf == true) //Nếu nút đó là lá (thì lặp lại 2
                     //việc sau:)
  {
    while (i >= 0 && keys[i] > k) //Tim vị trí thích
                                   //hợp có thể chèn khóa mới
    {
      ...
    }
  }
}

```

```

        keys[i+1] = keys[i];      //Chuyển tất cả các khóa
                                lớn hơn lên 1 vị trí phía trước
        i--;                  //(rồi) giảm chỉ số mảng xuống 1
    }

    keys[i+1] = k;           //Chèn khóa mới vào chỗ vừa
                            tìm được
    n = n+1;                //Cập nhật số khóa
    }

    else                   //Ngược lại, nếu nút không là lá (thì)
    {
        while (i >= 0 && keys[i] > k) //Tim con thích hợp
            để chèn khóa mới vào
        i--;
        if (C[i+1]->n == 2*d-1) //Nếu nút con đầy (thì)
        {
            splitChild(i+1, C[i+1]); //Tách nó thành 2 nút
            if (keys[i+1] < k)       //Nếu k lớn hơn khóa ở
                                nút con này (thì)
                i++;                //Tăng i lên một
            }
            C[i+1]->insertNonFull(k); ////(rồi) Chèn k vào nút
                                con đó
        }
    }
}

```

g. Hàm tìm khóa có giá trị nhỏ thua k cho trước

```

int BTREE::findKey(int k)
{
    int idx=0;
    while (idx<n && keys[idx] < k)
        ++idx;
    return idx;
}

```

h. Xóa một khóa

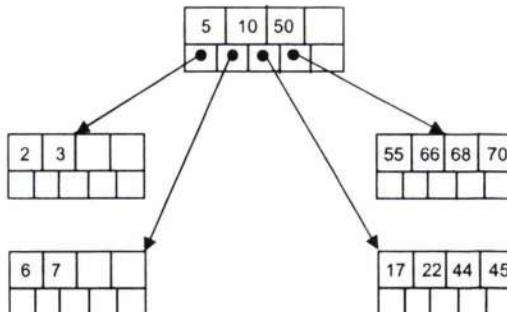
Mô tả thuật toán xóa bằng ngôn ngữ tự nhiên

Để xóa một khóa k trên B-cây, bắt đầu từ nút lá, có 2 bước sau:

1. Xóa k từ nút hiện hành.
2. Việc xóa k có thể sinh ra nút có quá ít giá trị.

Nếu một nút có quá ít giá trị ta gọi nó là nút dưới tràn (Underflowed). Nếu không có hiện tượng dưới tràn, ta sẽ tiến hành xóa khóa. Nếu có nút dưới tràn ta sẽ cố định (đánh dấu) nút ấy.

Làm cách nào để cố định một nút không phải là gốc mà nó bị dưới tràn? Ta xét ví dụ xóa khóa 6 từ B-cây có bậc 5 cho bởi hình 4.18:



Hình 4.18. Xóa giá trị 6

Việc xóa 6 sẽ gây ra nút dưới tràn, nút ấy chỉ chứa 1 giá trị (7). “Chiến thuật” của ta là “đánh dấu” nút này để cố “vay mượn” (borrow) giá trị từ một nút láng giềng. Ta sẽ “ghép nối” (gộp) nút hiện hành với một láng riêng “đóng đúc” (có nhiều giá trị) của nó, và trong nút đã được gộp này ta sẽ đặt một giá trị thích hợp (bảo toàn đặc tính của B-cây) lấy từ nút cha (nằm giữa 2 nút đang xét).

Trong ví dụ này, ta gộp nút [7] với láng giềng đóng đúc [17 22 44 45] và đặt giá trị 10 lấy từ cha (nằm giữa 2 nút đang xét) vào nút đã gộp để tạo thành nút [7 10 17 22 44 45].

Có bao nhiêu giá trị có thể đặt vào nút gộp? Câu trả lời là:

- Nút cha sẽ phân phát 1 giá trị xuống nút gộp.
- Nút dưới tràn được phân phát $((d-1)/2)-1$ giá trị.

- Nút láng giềng phân phát cho nút dưới trán một số giá trị trong khoảng $[(d-1)/2, (d-1)]$.

Việc xử lý nút gộp phụ thuộc khác nhau vào việc phân bổ $(d-1)/2$ giá trị hoặc hơn thế của nút láng giềng. Ta lần lượt tìm hiểu các trường hợp sau:

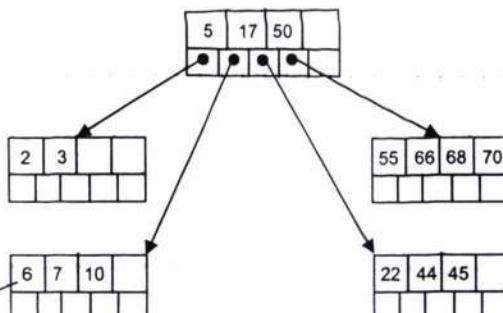
Trường hợp 1: Giả sử rằng nút láng giềng chứa hơn $(d-1)/2$ giá trị. Trong trường hợp này, tổng số các giá trị trong nút gộp lớn hơn $1 + ((d-1)/2 - 1) + ((d-1)/2)$, nghĩa là lớn hơn $(d-1)$. Tóm lại là nút này chứa d giá trị hoặc hơn.

Ta sẽ chia nút gộp thành 3 phần: trái, giữa, phải, trong đó phần giữa chỉ chứa 1 giá trị riêng lẻ ở vị trí thực sự giữa của nút gộp. Vì nút gộp có không dưới d giá trị, còn mỗi nút trái, phải mỗi nút bao đảm có $(d-1)/2$ giá trị và vì vậy các nút là hợp lệ. Ta sẽ thay thế giá trị “vay mượn” từ nút cha trước đây với nút giữa và biến nút trái, phải thành 2 nút con (của nút cha). Trong trường hợp này kích thước nút cha vẫn không thay đổi.

Như trên đã thấy, nút gộp $[7 \ 10 \ 17 \ 22 \ 44 \ 45]$ chứa hơn 5 giá trị (bị tràn), bởi vậy phải tách nó thành:

- Phần trái: $[7 \ 10]$
- Phần giữa: 17
- Phần phải: $[22 \ 44 \ 45]$

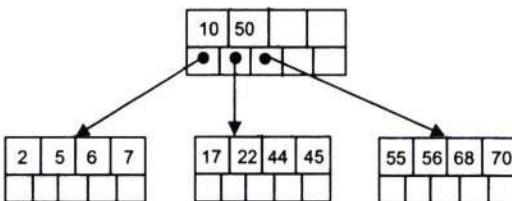
Ta đặt giá trị của phần giữa (17) vào đúng vị trí mà trước đây số 10 đã “đúng”. Còn các phần trái và phần phải trở thành nút con của gốc (là nút cha) (Chú ý: trong hình 4.18a) dưới đây không tồn tại 6 vì nó đã bị xóa rồi). Việc xóa giá trị 6 kết thúc!



Nhớ rằng khóa 6
không có trong hình này

Hình 4.18 a). Xóa giá trị 6

Trường hợp 2: Giả sử nút láng giềng chứa đúng $(d-1)/2$ giá trị. Vì vậy tổng số giá trị trong nút gộp là $1 + ((d-1)/2 - 1) + ((d-1)/2) = d-1$. Trong trường hợp này nút gộp chứa đúng số lượng cho phép các khóa (các giá trị) của B-cây bậc 5. Ta xét ví dụ sau để minh họa cho trường hợp này. Giả sử rằng ở B-cây đã cho, ta cần xóa khóa 3 thay vì xóa khóa 6. Khi xóa 3 khỏi nút [2 3] (xem hình 4.16), nút này vẫn là nút dưới tràn. Nó sẽ được kết hợp với nút láng giềng [6 7]. Trong nút gộp này xẽ xen thêm khóa 5 lấy từ vị trí cận trái của nút cha (ở đây là gốc của cây) và lúc này nút gộp là [2 5 6 7]. Nó chỉ có 4 khóa, nên ta không cần xử lý gì thêm. Kết quả xóa 3 sẽ được cây như hình 4.18 b):



Hình 4.18 b). Kết quả xóa khóa 3

Code C++ thiết kế hàm xóa khóa k từ cây con có gốc là nút hiện hành

```
void BTREE::remove(int k)
{
    int idx = findKey(k);           //Xem hàm này ở mục 5
                                    //trên đây
    if (idx < n && keys[idx] == k)   //Nếu chưa xét hết
        các nút và k cần xóa tồn tại ở vị trí idx (thì)
    {
        if (leaf) //Nếu nút hiện hành (nút ở idx) là
                    //lá (thì)
            removeFromLeaf(idx); //gọi hàm removeFromLeaf
                                    //để xóa khóa ở lá
        else
            removeFromNonLeaf(idx); /*Ngược lại gọi hàm
                                         removeFromNonLeaf để xóa khóa ở nút không là lá */
    }
    else
    {
```

```

if (leaf)      //Nếu nút hiện hành là lá thì in thông
               báo dưới đây:
{
    cout << " Khóa "<< k << " không tồn tại trên
               cây.\n";
    return;
}

bool flag = ( (idx==n)? true : false ); /*Cờ flag
báo hiệu "true" nếu khóa đã xóa ở cây con có gốc là nút
hiện hành, ngược lại là false*/
if (C[idx]->n < d)      //Nếu khóa ở trường n do con
               trỏ C[idx] trỏ đến nhỏ hơn số d các khóa
fill(idx);           //Khi ta sẽ làm đầy nút này
if (flag && idx > n) //Nếu flag cho trị true và chỉ
               số idx>n (thì)
    C[idx-1]->remove(k); //Gọi đệ quy xóa khóa ở
                           nút con thứ (idx-1)
else
    C[idx]->remove(k); //Ngược lại gọi đệ quy xóa
                           khóa ở nút con thứ idx
}
return;
}

Code C++ thiết kế hàm xóa khóa ở nút lá
void BTTreeNode::removeFromLeaf (int idx)
{
    for (int i=idx+1; i<n; ++i)
        keys[i-1] = keys[i]; /*Chuyển tất cả các khóa
               ở vị trí thứ i đến vị trí thứ i-1 (lùi lại phía trước 1 vị
               trí)*/
    n--;                      //Giảm số khóa đi 1
    return;
}

```

Chú ý: Những modul mô tả các thao tác còn lại (như: trộn (gộp, kết hợp) nút, vay mượn khóa, tách khóa, tìm tiền bối (getPrd)... đọc giả xem ở phần cài

đặt tổng thể dưới đây). Những nguyên mẫu (prototype) nào, những modul nào đã phân tích bằng tiếng Việt ở mục 4.3.5 sẽ không được nói lại ở phần cài đặt dưới đây.

4.3.6. Cài đặt B-cây trên C++

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>

class BTreenode
{
    int *keys;
    int d;
    BTreenode **C;
    int n;
    bool leaf;

public:
    BTreenode(int _d, bool _leaf);
    void traverse();
    BTreenode *search(int k);
    int findKey(int k);
    void insertNonFull(int k);
    void splitChild(int i, BTreenode *y);
    void remove(int k);
    void removeFromLeaf(int idx);
    void removeFromNonLeaf(int idx);
    int getPred(int idx);
    int getSucc(int idx);
    void fill(int idx);
    void borrowFromPrev(int idx);
    void borrowFromNext(int idx);
```

```
        void merge(int idx);
        friend class BTree;
    };

class BTree
{
    BTreeNode *root;           //Con trỏ nút gốc
    int d;
public:
    BTree(int _d)
    {
        root = NULL;
        d = _d;
    }
    void traverse()
    {
        if (root != NULL) root->traverse();
    }

    BTreeNode* search(int k)
    {
        return (root == NULL) ? NULL : root->search(k);
/*Nếu cây rỗng thì trả về NULL, ngược lại trả về khóa k
trên cây ấy*/
    }
    void insert(int k);
    void remove(int k);

};

BTreeNode::BTreeNode(int d1, bool leaf1) //Lưu bản
                                         sao của bậc và thuộc tính của lá
{
    d = d1;
```

```

if (C[idx]->n >= d)           //Nếu nút C[idx] có hơn d
                                //khóa (thì)
{
    int pred = getPred(idx);    //Tìm tiền bối "pred"
                                //của nó nhờ hàm getPred(idx)
    keys[idx] = pred;          //và thay k bởi pred rồi ghi
                                //nhớ pred đó tại idx (cập nhật k)
    C[idx]->remove(pred);    //Xóa nó tại idx nhờ gọi hàm
                                //remove(pred)

}

else if (C[idx+1]->n >= d)    //Còn nếu nút idx+1 có
                                //hơn d khóa (thì)
{
    int succ = getSucc(idx);   //Ta tìm hậu bối
                                //"succ" của nó và thay k bởi succ
    keys[idx] = succ;          //ghi nhớ (lưu) hậu bối này
    C[idx+1]->remove(succ);   //rồi xóa nó nhờ gọi hàm
                                //remove(succ)

}

//Còn nếu cả 2 nút C[idx] and C[idx+1] có ít hơn d
//khóa thì ta gộp (trộn) chúng lại
//bằng cách gộp tất cả k khóa của C[idx+1] vào C[idx],
lúc này C[idx] chứa 2d-1 khóa
//Giải phóng C[idx+1] và gọi hàm remove(k) xóa k khỏi
C[idx]

else
{
    merge(idx);
    C[idx]->remove(k);
}

return;
}

int BTreenode::getPred(int idx)      //Hàm tìm tiền bối
                                //của nút idx

```

```

{
    BTreenode *cur=C[idx]; //Cho con trỏ cur trỏ tới nút
                           hiện tại idx
    while(!cur->leaf) //Chúng nào mà nút hiện tại chưa
                        phải là lá
        cur = cur->C[cur->n]; //((thì) chúng đó ta còn di
                           chuyển con trỏ cur để
//Tim khóa cuối cùng của lá
    return cur->keys[cur->n-1]; //Trả về khóa cuối cùng
                                   của lá
}
int BTreenode::getSucc(int idx) //Hàm tìm hậu bối
                               của nút idx
{
    BTreenode *cur = C[idx+1]; //Cho con trỏ cur trỏ
                               tới nút hiện tại idx+1
    while (!cur->leaf) //Chúng nào nút hiện tại
                        chưa phải là lá
        cur = cur->C[0]; //((thì) chúng đó ta còn tìm khóa
                           đầu tiên của lá
    return cur->keys[0]; //Trả về khóa đầu tiên của lá
                           khi tìm thấy nút này
}
void BTreenode::fill(int idx) //Hàm lấp đầy nút con
                           C[idx] khi nó có ít hơn d-1 khóa
{
    if (idx!=0 && C[idx-1]->n>=d) //Nếu nút con
        trước C[idx-1] có nhiều hơn d-1 khóa, ta sẽ
        borrowFromPrev(idx); //Vay (mượn) một khóa ở đó
    else if (idx!=n && C[idx+1]->n>=d) //Còn nếu nút
        kế sau C[idx+1] có nhiều hơn d-1 khóa
        borrowFromNext(idx); //((thì) Ta sẽ vay một khóa
                           ở nút này
    else //Còn thi làm:
    {
        if (idx != n) //Nếu C[idx] không là con
                      cuối cùng thì gộp nó với nút anh-em kế sau nó

```

```

        merge(idx);
    else//Ngược lại thì gộp nó với anh em đứng trước nó
        merge(idx-1);
    }
    return;
}

void BTreenode::borrowFromPrev(int idx) //Hàm vay một
                                         //khóa ở C[idx-1] rồi chèn vào C[idx]
{
    BTreenode *child=C[idx];
    BTreenode *sibling=C[idx-1];
    //Khóa cuối cùng từ nút C[idx-1] đưa lên nút cha
    //và khóa key[idx-1] từ nút cha
    //được chèn vào nút C[idx] như là khóa đầu tiên.
    Việc này làm biến mất một khóa anh em và lại xuất hiện một
    khóa con.

    for (int i=child->n-1; i>=0; --i)
        child->keys[i+1] = child->keys[i]; //Chuyển tất
        //cả các khóa ở nút C[idx] lên phía trước 1 vị trí
    if (!child->leaf) //Nếu C[idx] không là lá thì
        //chuyển tất cả con của nó lên phía trước 1 vị trí
    {
        for(int i=child->n; i>=0; --i)
            child->C[i+1] = child->C[i];
    }

    child->keys[0] = keys[idx-1]; /* Đặt khóa thứ nhất
    của nút con bằng khóa key[idx-1] từ nút hiện hành */
    if (!leaf) //Nếu nút hiện hành không phải lá
        //thì chuyển
    child->C[0] = sibling->C[sibling->n]; //con cuối
        //của nút anh em thành con đầu của C[idx]
    keys[idx-1] = sibling->keys[sibling->n-1]; //Chuyển
        //khóa từ nút anh em lên nút cha
    child->n += 1; //Tăng số khóa của nút con lên 1
}

```

```

    sibling->n -= 1; //Giảm số khóa của anh em xuống 1
    return;
}
void BTreenode::borrowFromNext(int idx) /*Hàm vay một
khóa từ nút C[idx+1] rồi đặt nó vào nút C[idx] */
{
    BTreenode *child=C[idx];
    BTreenode *sibling=C[idx+1];
    child->keys[(child->n)] = keys[idx];/*Khóa keys[idx]
được chèn vào C[idx] là khóa cuối cùng của nút này*/
    if (!(child->leaf)) /*Nếu nút con hiện hành không
phải lá (thì) nút con thứ nhất của nút anh em được chèn
vào C[idx] như là con cuối cùng của nút C[idx]*/
        child->C[(child->n)+1] = sibling->C[0];
    keys[idx] = sibling->keys[0];           //Khóa đầu tiên từ
                                            nút anh em được chèn vào keys[idx]
    for (int i=1; i<sibling->n; ++i)
        sibling->keys[i-1] = sibling->keys[i]; //Chuyển tất
                                                cả các khóa của nút anh em về phía sau 1 vị trí
    if (!sibling->leaf) //Nếu nút anh em không phải là
                          nút lá (thì)
    {
        for(int i=1; i<=sibling->n; ++i)
            sibling->C[i-1] = sibling->C[i]; //Chuyển các
                                                con trỏ nút về phía sau một vị trí
    }
    child->n += 1; //Tăng số lượng khóa của nút C[idx]
(nút con) lên 1
    sibling->n -= 1; //Giảm số lượng khóa của nút
                      C[idx+1] (nút anh em) xuống 1
    return;
}

void BTreenode::merge(int idx) //Hàm gộp (trộn) C[idx]
                           với C[idx+1]

```

```

        //Sau trộn C[idx+1] được giải phóng
        BTreenode *child = C[idx];
        BTreenode *sibling = C[idx+1];

        child->keys[d-1] = keys[idx];      /*Kéo một khóa từ
nút hiện hành và chèn nó vào vị trí thứ (d-1) trong nút
C[idx]*/
        for (int i=0; i<sibling->n; ++i)
            child->keys[i+d] = sibling->keys[i];    //Copy các
khóa từ C[idx+1] đến vị trí cuối ở C[idx]
        if (!child->leaf) //Nếu nút con không phải lá (thì)
        {
            for(int i=0; i<=sibling->n; ++i)
                child->C[i+d] = sibling->C[i];    //Copy các con trỏ
nút từ C[i] đến C[i+d]
        }
        for (int i=idx+1; i<n; ++i)
            keys[i-1] = keys[i];    //Chuyển tất cả các khóa ở
sau idx trong nút hiện hành lên phía trước 1 vị trí
        for (int i=idx+2; i<=n; ++i)
            C[i-1] = C[i];    //Chuyển các con trỏ sau vị trí
(idx+1) ở nút hiện hành lên phía trước 1 vị trí
        child->n += sibling->n+1;    //Cập nhật số khóa của
nút con
        n--;    //Và cập nhật số khóa ở nút hiện hành
        delete(sibling);    //Giải phóng vùng nhớ đã cấp cho
nút anh em
        return;
    }

Void BTree::insert(int k) //Hàm chính để chèn một khóa
mới vào B-cây
{
    if (root == NULL)           //Nếu cây rỗng (thì)
    {

```

```

root = new BTTreeNode(t, true);      //Cấp phát vùng
                                    nhớ cho gốc
root->keys[0] = k;    //Chèn khóa k vào vị trí đầu
root->n = 1;          //Ghi nhớ số khóa ở gốc
}
else           //Ngược lại nếu cây không rỗng (thì)
{
    if (root->n == 2*d-1) //Nếu nút gốc đầy, phải
                            tăng chiều cao cây
    {
        BTTreeNode *s = new BTTreeNode(t, false); //Bằng
                                    cách cấp phát vùng nhớ cho gốc mới

        s->C[0] = root; //Thiết lập gốc cũ là con của
                           gốc mới
        s->splitChild(0, root); //Tách nút cũ và
                                chuyển một khóa tới gốc mới
                                //Bây giờ gốc mới có 2 con
        int i = 0;
        if (s->keys[0] < k) //Nếu khóa ở gốc < k (thì)
            i++;           //Tăng chỉ số i lên một
        s->C[i]->insertNonFull(k); //Gọi hàm
                                    insertNonFull để chèn k vào nút s
        root = s;           //((rồi) Đổi gốc
    }
    else           //Còn nếu gốc không đầy thì
        root->insertNonFull(k); //gọi hàm
                                    insertNonFull để chèn k vào gốc
    }
}

void BTTreeNode::insertNonFull(int k)
{
    int i = n-1; //Khởi trị cho chỉ số là chỉ số của
                  phần tử phải nhất (cận phải)
}

```

```

if (leaf == true) //Nếu phần tử này là nút lá (thì)
    lặp lại 2 việc sau:
{
    // a) Tìm vị trí thích hợp để chèn khóa mới
    // b) Chuyển tất cả các khóa lớn hơn lên một vị
trí ở phía trước

    while (i >= 0 && keys[i] > k) //Chỉ cần i ≥ 0
        đồng thời keys[i]>k (thì)
    {
        keys[i+1] = keys[i]; //Lặp lại việc chuyển khóa
        ở i lên i+1 (ở phía trước)
        i--; //Giảm chỉ số i xuống 1 (vì ta chuyển khóa
        về phía trước 1 vị trí)
    }
    keys[i+1] = k; //Chèn khóa mới vào vị trí
    tìm được
    n = n+1; //Cập nhật số lượng khóa
}

else //Còn nếu nút đó không phải là lá (thì)
    ta tìm nút con mà nó sẽ có khóa mới
    while (i >= 0 && keys[i] > k) //Chỉ cần i ≥ 0 đồng thời keys[i]>k (thì) lặp:
    i--; //Giảm chỉ số i
    if (C[i+1]->n == 2*d-1) //Nếu nút con tìm thấy
        là nút đầy
    {
        splitChild(i+1, C[i+1]); //Thì Tách nút ấy
        if (keys[i+1] < k) //Nếu khóa ở (i+1) nhỏ hơn
            k (thì)
            i++; //Tăng i lên một đơn vị
    }
    C[i+1]->insertNonFull(k); //rồi Gọi hàm
    insertNonFull(k) để chèn k vào chỗ đó
}
}

```

```

void BTreenode::splitChild(int i, BTreenode *y) //Hàm
    tách nút con y của nút hiện tại
{
    BTreenode *z = new BTreenode(y->d, y->leaf); //Tạo
    nút mới để
    z->n = d - 1; //Lưu d-1 khóa của y vào trường n do z
    trở tới
    for (int j = 0; j < d-1; j++)
        z->keys[j] = y->keys[j+d]; //Copy d-1 khóa cuối
    cùng của y vào z
    if (y->leaf == false) //Nếu y không phải
    là lá (thì)
    {
        for (int j = 0; j < d; j++)
            z->C[j] = y->C[j+d]; //Gán phần tử C[j+d] của
        nút y cho phần tử C[j] của z
    }
    y->n = d - 1; //Giảm số lượng khóa của y,
    nút này sẽ có con mới
    for (int j = n; j >= i+1; j--)
        C[j+1] = C[j]; //Tạo một không gian (chỗ)
    cho nút con mới (của y)
        C[i+1] = z; //Liên kết nút con mới này
    với nút hiện hành
//Một khóa của y chuyển đến nút hiện hành, ta tìm chỗ cho
khóa mới và
    for (int j = n-1; j >= i; j--)
        keys[j+1] = keys[j]; //Chuyển tất cả các khóa
    lớn hơn lên phía trước 1 vị trí
        keys[i] = y->keys[d-1]; //Copy khóa giữa của y
    đến nút i
        n = n + 1; //Cập nhật số khóa của nút này
    }
void BTreenode::traverse() //Hàm duyệt tất cả
    các nút của cây con có gốc là nút đang xét
}

```



```

cout << "Cây rỗng \n";           //Thông báo cây rỗng
return;
}
root->remove(k);      //Gọi hàm remove để xóa khóa k
if (root->n==0)        //Nếu gốc không có khóa, thì
                        tạo con đầu tiên là gốc mới
{
    BTreenode *tmp = root;
    if (root->leaf)          //Nếu gốc là lá
        root = NULL;         //Thì đặt gốc là NULL
    else
        root = root->C[0];   //Ngược lại gốc là nút
                                con đầu tiên
    delete tmp;             //Giải phóng nút gốc cũ
}
return;
}

//Dưới đây là trình điều khiển để kiểm thử (Test) một số
hàm đã thiết kế trên đây
int main()
{
    BTreenode t(3);           //B-cây có bậc 3

    t.insert(1);
    t.insert(3);
    t.insert(7);
    t.insert(10);
    t.insert(11);
    t.insert(13);
    t.insert(14);
    t.insert(15);
    t.insert(18);
}

```

```
t.insert(16);
t.insert(19);
t.insert(24);
t.insert(25);
t.insert(26);
t.insert(21);
t.insert(4);
t.insert(5);
t.insert(20);
t.insert(22);
t.insert(2);
t.insert(17);
t.insert(12);
t.insert(6);

cout << "Duyệt cây đã được tạo:\n";
t.traverse();
cout << endl;
t.remove(6);
cout << "Duyệt cây sau khi xóa 6:\n";
t.traverse();
cout << endl;

t.remove(13);
cout << " Duyệt cây sau khi xóa 13:\n";
t.traverse();
cout << endl;
t.remove(7);
cout << " Duyệt cây sau khi xóa 7:\n";
t.traverse();
cout << endl;
```

```

t.remove(4);
cout << "Duyệt cây sau khi xóa 4:\n";
t.traverse();
cout << endl;

t.remove(2);
cout << "Duyệt cây sau khi xóa 2:\n";
t.traverse();
cout << endl;

t.remove(16);
cout << "Duyệt cây sau khi xóa 16:\n";
t.traverse();
cout << endl;

return 0;
}

```

4.3.7. Đánh giá hiệu quả của B-cây

Cũng như các thuật toán cho bộ nhớ ngoài khác, tham số quan trọng nhất cho B-cây không là tổng thời gian tính toán, mà là số lần truy cập bộ nhớ. Số lần truy cập bộ nhớ trong mỗi thao tác trên B-cây tỉ lệ với chiều cao của cây.

b. Trường hợp tốt nhất

Gọi h là chiều cao của B-cây; gọi n là số các mục vào trên cây này; còn m là số tối đa các cây con của một nút. Mỗi nút có nhiều nhất $m-1$ khóa. Bằng quy nạp ta có thể thấy rằng B-cây với chiều cao h và tất cả các khóa đều được lắp hoàn toàn vào các nút thì cây đó có $n=(mh-1)$ khóa. Từ đó suy ra trường hợp tốt nhất chiều cao h của B-cây được đánh giá bởi (*) dưới đây:

$$h = \lceil \log_m(n+1) \rceil \quad (*)$$

Gọi $d = \lceil m/2 \rceil$ là số tối thiểu các cây con của một nút nội (nhưng không là gốc). Ta xét trường hợp tồi nhất sau đây:

b. Trường hợp tồi nhất

Theo Cormen Thomas H.; Leiserson Charles E. và... trong tài liệu [5], một B-cây với n nút, trong trường hợp tồi nhất, có chiều cao h đánh giá bởi hệ thức (*) dưới đây:

$$h \leq \left\lceil \log_d \left(\frac{n+1}{2} \right) \right\rceil \quad (*) \quad \text{Trong đó } d = [m/2]$$

4.3.8. Tính hiệu quả của B-cây với cách đánh giá khác

Do có nhiều bản ghi trên một nút và nhiều nút trên cùng một mức vì dữ liệu được lưu trữ trên đĩa nên ta cần các thao tác trên B-cây càng nhanh càng tốt. Giả sử có một danh bạ điện thoại có 500.000 bản ghi (record, structure). Tất cả các nút trong B-cây đều có ít nhất một nửa là nút đầy, vì thế chúng chưa đựng ít nhất 8 bản ghi và 9 liên kết đến các nút con. Kết quả là chiều cao của cây nhỏ hơn $\log_9 N$ (logarit cơ số 9 của N), trong đó N là 500.000. Logarit này sẽ có giá trị là 5.972, do đó cây sẽ có 6 mức.

Kết quả là, khi sử dụng B-cây chỉ cần 6 lần truy cập đĩa để tìm bất kỳ bản ghi nào trong tệp tin chứa 500.000 bản ghi. Với 10 mili giây trên một lần truy cập, sẽ chiếm khoảng 60 mili giây hay 6/100 giây. Kết quả này nhanh hơn rất nhiều so với việc tìm kiếm nhị phân trong tệp tin có thứ tự tuần tự.

Càng nhiều bản ghi trong một nút, cây sẽ có càng ít các mức. Như chúng ta đã thấy ở ví dụ trên, chỉ cần 6 mức cho 500.000 bản ghi, dù mỗi nút chỉ lưu trữ có 16 bản ghi. Ngược lại, trong cây nhị phân với 500.000 mục dữ liệu sẽ cần khoảng 19 mức, còn trong cây 2-3-4 sẽ cần 10 mức. Nếu ta sử dụng các khối với hàng trăm bản ghi, ta có thể giảm số lượng các mức trong cây xuống và cải tiến được số lần truy cập đĩa.

Không những việc tìm kiếm trên B-cây nhanh hơn so với trong tệp tin có thứ tự trên đĩa, mà đối với việc chèn và xóa ở B-cây cũng hiệu quả hơn rất nhiều.

Việc chèn khóa mới vào B-cây không cần phải tách nút. Đây là tính hướng thông dụng nhất, bởi vì một số lượng lớn các bản ghi nằm trên một nút. Trong ví dụ danh bạ điện thoại chỉ cần 6 lần truy cập để tìm kiếm điểm chèn. Khi đó sẽ có nhiều hơn một lần truy cập để ghi khỏi có chứa bản ghi mới cần chèn vào đĩa. Tổng cộng có 7 lần truy cập.

Xét việc tách nút: Nút bị tách phải được đọc (có một nửa các bản ghi của nó bị xoá đi) và ghi lại đĩa. Nút mới tạo ra phải được ghi vào đĩa và nút cha của nó phải được đọc (tương tự như việc chèn bản ghi nâng lên mức trên) và ghi lại đĩa. Điều này cần thêm 5 lần truy cập cộng với 7 lần truy cập để tìm kiếm điểm chèn, tổng cộng ta có 12 lần. Đây là sự cải tiến rất đáng kể so với 500.000 lần truy cập cho việc chèn vào tệp tin tuần tự.

Trong các B-cây cải tiến (B+, B*), chỉ có các node lá mới chứa các bản ghi. Các nút không phải là lá chỉ chứa các khóa và các số của khối. Điều này có thể thực hiện thao tác nhanh hơn bởi vì mỗi khối có thể lưu nhiều hơn các số của khối. Kết quả là cây có bậc cao hơn sẽ có ít mức hơn và tốc độ truy cập sẽ được tăng lên. Tuy nhiên, việc lập trình sẽ trở nên phức tạp bởi vì có 2 loại nút: nút lá và nút không phải lá.

4.3.9. Tệp tin chỉ mục

Một cách tiếp cận khác để tăng tốc độ truy cập tệp tin là lưu các bản ghi với thứ tự tuần tự và sử dụng một tệp tin chỉ mục (Index File) ứng với dữ liệu của chính nó. Một tệp tin chỉ mục là danh sách các cặp key/block (khóa/khối), được sắp theo thứ tự của key. Xem lại ví dụ danh bạ điện thoại chúng ta có 500.000 bản ghi với mỗi bản ghi cần 512 byte, một khối lưu 16 bản ghi và có 31.250 khối. Giả sử rằng khóa tìm kiếm là họ, mỗi đầu vào của chỉ mục chứa đựng 2 mục:

- Khóa, chẳng hạn như Jones, và

- Số hiệu của khối chứa đựng bản ghi Jones trong tệp tin. Các số này chạy từ 0 đến 31.249.

Giả sử chúng ta sử dụng một xâu (string) có chiều dài 28 byte cho mỗi khóa (đủ cho hầu hết các họ) và 4 byte cho số hiệu của khối (là kiểu int trong Java). Kết quả là mỗi đầu vào trong chỉ mục của ta sẽ chiếm khoảng 32 byte. Chỉ chiếm 1/16 số lượng cần thiết đối với mỗi bản ghi.

Các đầu vào của chỉ mục được sắp xếp theo họ. Các bản ghi gốc trên đĩa có thể được sắp với bất kì thứ tự quy ước nào. Điều này có nghĩa là các bản ghi mới đơn giản chỉ cần thêm vào cuối tệp tin, vì thế các bản ghi có thứ tự theo thời gian chèn vào. Sự sắp xếp này được trình bày trong hình 4.19.

Khối #2385						Khối #2386					
0	1			15		0			15		
Jones	BenBrenh			Keler	Parker				Abbot	Duncan	

Khóa	Khối số
Jone	2385
Jordan	5471
Joslyn	1802
Joyce	7429
Jung	3423

512 Bytes

32 bytes

Hình 4.19. Tệp chỉ mục

4.3.10. Tập tin chỉ mục trong bộ nhớ

Vì các tập tin chỉ mục nhỏ hơn nhiều so với các tập tin chứa đựng các bản ghi thực sự, nên có thể làm cho tập tin chỉ mục đủ nhỏ để đưa toàn bộ vào bộ nhớ chính. Trong ví dụ trên có 500.000 bản ghi. Mỗi bản ghi có đầu vào 32 byte trong tập tin chỉ mục, vì thế tập tin chỉ mục sẽ có $32 * 500.000$ hay 1.600.000 byte (1,6 MB). Trong các máy tính hiện đại, không có vấn đề trờ ngại nào về lưu trữ trong bộ nhớ RAM với kích cỡ 1,6 MB. Tập tin chỉ mục có thể lưu trữ trên đĩa, nhưng nó sẽ được đọc vào bộ nhớ bất cứ khi nào chương trình cơ sở dữ liệu khởi động. Từ trước đến nay, các thao tác trên tập tin chỉ mục có thể tiến hành trong bộ nhớ. Thường xuyên tập tin chỉ mục có thể ghi lại lên đĩa để đáp ứng lưu trữ lâu dài và phòng vé.

a. Tìm kiếm trên các tập chỉ mục

Cách tiếp cận tập tin chỉ mục trong bộ nhớ cho phép các thao tác thực hiện trên tập tin danh bạ điện thoại nhanh hơn so với tập tin mà các bản ghi được sắp theo thứ tự nào đó. Ví dụ, tìm kiếm nhị phân cần 19 lần truy cập tập tin chỉ mục. Với 20 micro giây trên một lần truy cập chỉ cần khoảng 4/10.000 giây. Sau đó có lẽ sẽ có thời gian để đọc bản ghi thực sự từ đĩa (khi mà số hiệu của khối đã được tìm thấy trong tập tin chỉ mục). Tuy nhiên, điều này chỉ cần một lần truy cập đĩa, chiếm khoảng 10 mili giây.

b. Thao tác chèn trên các tập chỉ mục

Để chèn một mục mới vào tập tin đã có chỉ mục, cần thiết phải thông qua 2 bước. Trước tiên chúng ta chèn bản ghi đầy đủ của nó vào tập tin chính; sau đó chúng ta chèn một đầu vào, chứa đựng khóa và số hiệu của khối mà bản ghi

mới sẽ được lưu trữ vào tệp tin chỉ mục. Bởi vì tệp tin chỉ mục là có thứ tự tuần tự, để chèn một mục mới chúng ta cần phải di chuyển trung bình một nửa các đầu vào của tệp tin chỉ mục. Cần 2 micro giây để di chuyển một byte trong bộ nhớ, khi đó chúng ta cần $250.000 * 32 * 2$ micro giây, hay khoảng 16 giây để chèn một đầu vào mới. So với 5 phút đối với tệp tin tuần tự không có chỉ mục thì đó là một tốc độ chèn rất cao (chú ý rằng chúng ta không cần phải dịch chuyển bắt kí một bản ghi nào trong tệp tin chính. Đơn giản là ta chỉ thêm bản ghi mới vào cuối của tập tin).

Có thể sử dụng cách tiếp cận tinh vi hơn để lưu trữ tệp tin chỉ mục trong bộ nhớ. Chẳng hạn cây nhị phân, cây 2-3-4, cây đũa đen. Bất kì cách lưu trữ nào ở trên cũng giảm thời gian đáng kể cho việc chèn và xóa. Trong bất cứ trường hợp nào cách tiếp cận tệp tin chỉ mục trong bộ nhớ đều nhanh hơn cách tiếp cận tệp tin tuần tự. Trong một số trường hợp có thể nhanh hơn B-tree.

Việc truy cập đĩa thực sự cần thiết cho việc chèn vào tệp tin đã có chỉ mục một bản ghi mới. Thông thường khôi trước đó của tệp tin được đọc vào bộ nhớ, bản ghi mới được thêm vào đây, sau đó khôi này được ghi ra đĩa trở lại. Điều này chỉ cần 2 lần truy cập tệp tin.

4.3.11. Đa chỉ mục

Một thuận lợi nữa của cách tiếp cận chỉ mục là đa chỉ mục (mỗi chỉ mục là một khóa khác nhau) có thể được tạo cho cùng một tệp tin. Với một chỉ mục, các khóa có thể là họ, số điện thoại, địa chỉ. Bởi vì các chỉ mục là khá nhỏ so với tệp tin, điều này sẽ không tăng nhiều lâm lượng dữ liệu lưu trữ. Tất nhiên, sẽ có nhiều thử thách hơn khi các mục dữ liệu bị xóa khỏi tệp tin, bởi vì các đầu vào phải được xóa khỏi các chỉ mục, ở đây chúng ta không đi sâu vào vấn đề này.

4.3.12. Tệp tin chỉ mục quá lớn đối với bộ nhớ

Nếu tệp tin chỉ mục quá lớn để đưa vào bộ nhớ, nó phải được phân ra thành các khôi và lưu trữ trên đĩa. Đối với các tệp tin lớn, rất hiệu quả khi lưu trữ tệp tin chỉ mục như là B-tree. Trong tệp tin chính các bản ghi được lưu trữ theo bất kì thứ tự quy ước nào.

Cách sắp đặt này rất hiệu quả. Việc thêm các bản ghi vào cuối của tệp tin chính là một thao tác nhanh và đầu vào của tệp tin chỉ mục đổi với bản ghi mới cũng nhanh chóng để chèn vào bởi vì tệp tin chỉ mục là một cây. Kết quả là việc tìm kiếm và việc chèn vào rất nhanh chóng đối với các tệp tin lớn.

Chú ý: Khi tập tin chỉ mục được tổ chức như một B-tree, mỗi nút sẽ chứa số lượng các con trỏ đến các nút con của nó và một ít các mục dữ liệu. Các con trỏ con là số hiệu khôi của các nút khác trong tập tin chỉ mục. Còn các mục dữ liệu thì chứa một giá trị khóa và một con trỏ đến khôi trong tập tin chính. Không nên nhầm lẫn hai loại con trỏ khôi này.

4.3.13. Sơ lược về các tiêu chuẩn tìm kiếm phức tạp

Trong tìm kiếm phức tạp chỉ có cách tiếp cận thực tế là đọc mỗi khôi của tập tin một cách tuần tự. Giả sử trong ví dụ danh bạ điện thoại, chúng ta muốn liệt kê tất cả các đầu vào trong danh bạ có tên là Frank sống ở Springfield và có số điện thoại với ba số "7" (các thông tin này là đầu mối tìm kiếm...). Một tập tin được tổ chức theo họ sẽ không hiệu quả. Thậm chí có các tập tin chỉ mục được xếp thứ tự bởi tên và thành phố, sẽ không có cách quy ước nào để tìm các tập tin chứa đựng cả hai từ Frank và Springfield. Trong các trường hợp như vậy (khá thông dụng trong nhiều loại cơ sở dữ liệu) cách tiếp cận nhanh nhất là *đọc tập tin một cách tuần tự, từng khôi một*, tìm kiếm trên mỗi bản ghi để nhận biết bản ghi thỏa mãn tiêu chuẩn cần tìm.

TỪ VIẾT TẮT

Từ viết tắt	Tiếng Anh	Tiếng Việt
ATM	Asynchronous Transfer Mode	Ché độ truyền không đồng bộ
BMM	Balanced Multiway Merging	Sắp xếp trộn đa đường cân bằng
BST	Binary Search Tree	Cây tìm kiếm nhị phân
CCM	Coalesced Chaining Method	Phương pháp kết nối hợp nhất
DCM	Direct Chaining Method	Phương pháp kết nối trực tiếp
DHM	Double Hashing Method	Phương pháp băm kép
DPT		Độ phức tạp
DSLKD		Danh sách liên kết đơn
LPM	Linear Probing Method	Phương pháp dò tuyến tính
LR	Left – Rotate	Phép quay trái
MST	Multiway Search Tree	Cây đa nhánh tìm kiếm
PMS	Polyphase Merge Sort	Trộn đa pha
QPM	Quadric Probing Method	Phương pháp dò bậc hai
RBT	Red-Black Tree	Cây đỏ đen
RR	Right – Rotate	Phép quay phải

TÀI LIỆU THAM KHẢO

1. Clifford A. Shaffer, *Data Structures and Algorithm Analysis in C++*, Edition 3rd (C++ Version). Department of Computer Science Virginia Tech, Blacksburg, VA 24061, September 15, 2011, Copyright © 2009-2011 by Clifford A. Shaffer.
2. Mark Allen Weiss, *Data Structures & Algorithm Analysis in C++*. Second Edition. Addison-Wesley, 1999.
3. Frank M.Carrano, Janet J.Prichard, *Data Abstraction and Problem Solving with C++*. Third Edition. Addison-Wesley, 2010.
4. Alfred V. Aho, J.E. Hopcroft, Jeffrey D. Ullman, *Data Structures and Algorithms*, Addison-Wesley Series in Computer Science and Information Processing, 1983.
5. Cormen Thomas H.; Leiserson Charles E.; Rivest Ronald L.; Stein Clifford, *Introduction to Algorithms*. 3rd Ed, 2009. MIT Press and McGraw-Hill.
6. William H.Ford, William R.Topp, *Data Structures With C++ Using STL*. Publisher Prentice Hall, 2 nd, 2001.
7. Bayer Rudolf (1971), *Binary B-Trees for Virtual Memory*, Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, California.
8. Peter Brass, *Advanced Data Structures*, Cambridge University Press, 2008.
9. Robert Sedgewick and Kevin Wayne, Princeton University (USA), *Algorithms*. 4th Edition. Addition-Wesley Publishing Co, 2012.
10. Donald Knuth, *The Art of Computer Programming*, vol. 1,2,3. Addison-Wesley, 3rd edition, 1997.
11. Dinesh Mehta and Sartaj Sahni, *Handbook of Data Structures and Applications*, Chapman and Hall/CRC Press, 2007.
12. Niklaus Wirth, *Algorithms and Data Structures*. Prentice Hall, 1985.

13. Bayer Rudolf; McCreight E. (1970). *Organization and Maintenance of Large Ordered Indices, Mathematical and Information Sciences Report No. 20*. Boeing Scientific Research Laboratories.
14. Truong Hải Bằng. *Cấu trúc dữ liệu 2*. Nhà xuất bản Đại học Quốc gia TP. Hồ Chí Minh. 2007.

MỤC LỤC

<i>Lời nhà xuất bản</i>	3
<i>Lời nói đầu</i>	5
Chương 1: SẮP XÉP NGOÀI	7
1.1. Phương pháp xếp trộn các run	7
1.1.1. Một số khái niệm cơ bản về run	7
1.1.2. Thuật toán xếp trộn hai run	7
1.1.3. Thuật toán trộn tự nhiên	12
1.2. Thuật toán sắp xếp trộn đa đường cân bằng	19
1.2.1. Mô tả ngắn gọn các bước của BMM bằng ngôn ngữ tự nhiên	20
1.2.2. Mô tả chi tiết thuật toán BMM bằng mã giả	21
1.2.3. Cài đặt	24
1.3. Thuật toán trộn đa pha	24
1.3.1. Mở đầu	24
1.3.2. Mô tả thuật toán PMS bằng ngôn ngữ tự nhiên	24
1.3.3. Ví dụ cách cải tiến của PMS nhờ dùng hệ thức (F)	25
Chương 2: PHÉP BĂM VÀ BẢNG BĂM	26
2.1. Các khái niệm cơ bản	26
2.1.1. Phép băm hay hàm băm	26
2.1.2. Bảng băm	26
2.1.3. Các chỉ tiêu về một hàm băm tốt	27
2.1.4. Đúng đẽ trong bảng băm là gì?	27
2.1.5. Một vài loại hàm băm điển hình	28

<i>2.2. Các phương pháp giải quyết định độ trong bảng băm</i>	32
2.2.1. Tạo bảng băm bằng phương pháp kết nối trực tiếp	33
2.2.2. Tạo bảng băm bằng phương pháp kết nối hợp nhất	42
2.2.3. Tạo bảng băm dùng phương pháp dò tuyến tính	51
2.2.4. Tạo bảng băm dùng phương pháp dò bậc hai	59
2.2.5. Tạo bảng băm dùng phương pháp băm kép	63
<i>2.3. So sánh các phương pháp giải quyết định độ bằng số bước dò và hệ số nạp λ</i>	68
Chương 3: CÂY ĐỎ ĐEN	69
3.1. Mở đầu	69
3.2. Các khái niệm cơ bản về cây đỏ đen	69
3.2.1. Định nghĩa	69
3.2.2. Các đặc điểm (các luật) của RBT	69
3.2.3. Các tính chất của RBT	70
3.3. Các thao tác cơ bản trên RBT	72
3.3.1. Phép quay	72
3.3.2. Phép chèn	74
3.3.3. Phép xóa	79
3.4. Cài đặt trên C ⁺⁺	86
3.5. Đánh giá hiệu quả của RBT	104
Chương 4: CÂY 2-3-4 VÀ B-CÂY	106
4.1. Cây 2-3-4	106
4.1.1. Mở đầu	106
4.1.2. Tổ chức cây 2-3-4	106
4.1.3. Các phép biến đổi không làm thay đổi tính chất của cây 2-3-4.....	108

4.1.4. Mã giả mô tả một số phép cơ bản trên cây 2-3-4	114
4.1.5. Đánh giá tính hiệu quả của cây 2-3-4.....	121
4.2. Cây đa nhánh tìm kiếm	122
4.2.1. Định nghĩa cây đa nhánh tìm kiếm	122
4.2.2. Các tính chất của MST	122
4.3. B-Cây	124
4.3.1. Nguồn gốc chữ "B" trong tên gọi B-cây (B-Tree).....	124
4.3.2. Giới thiệu chung về B-cây	124
4.3.3. Định nghĩa B-cây	125
4.3.4. Các biến thể (Variants) của B-cây	126
4.3.5. Mã mô tả các thao tác trên B-cây.....	126
4.3.6. Cài đặt B-cây trên C ⁺⁺	140
4.3.7. Đánh giá hiệu quả của B-cây	155
4.3.8. Tính hiệu quả của B-cây với cách đánh giá khác	156
4.3.9. Tệp tin chỉ mục.....	157
4.3.10. Tệp tin chỉ mục trong bộ nhớ	158
4.3.11. Đa chỉ mục	159
4.3.12. Tệp tin chỉ mục quá lớn đối với bộ nhớ	159
4.3.13. Sơ lược về các tiêu chuẩn tìm kiếm phức tạp	160
Từ viết tắt	161
Tài liệu tham khảo	162

CẤU TRÚC DỮ LIỆU VÀ THUẬT TỐÁN

Chịu trách nhiệm xuất bản, nội dung

Giám đốc - Tổng Biên tập

TRẦN CHÍ ĐẠT

Biên tập:

NGUYỄN LONG BIÊN TRƯƠNG MINH ĐỨC

Trình bày sách:

TRƯỜNG ĐỨC LỢI

Sửa bản in:

TRƯỜNG MINH DỨC

Trình bày bài:

TRẦN HỒNG MINH

NHÀ XUẤT BẢN THÔNG TIN VÀ TRUYỀN THÔNG

Website: www.nxbthongtintruyenthong.vn; book365.vn; ebook365.vn

Trụ sở: Tầng 6, Tòa nhà Cục Tần số vô tuyến điện, số 115 đường Trần Duy Hưng, quận Cầu Giấy, TP. Hà Nội

ĐT Biên tập: 024.35772143 ĐT Phát hành: 024.35772138

E-mail: nxb.tttt@mic.gov.vn Fax: 024.35579858

Chi nhánh TP. Hồ Chí Minh: 8A đường D2, P25, quận Bình Thạnh, TP. Hồ Chí Minh.

Điện thoại: 028.35127750, 08.35127751 Fax: 028.35127751

E-mail: cnsq.nxbttt@mic.gov.vn

Chi nhánh TP. Đà Nẵng: 42 Trần Quốc Toản, quận Hải Châu, TP. Đà Nẵng

Điện thoại: 0236.3897467 Fax: 0236.3843359

E-mail: cndn.nxbttt@mic.gov.vn

Chi nhánh Tây Nguyên: 28B, Y Bih Alêô, TP. Buôn Ma Thuột, Tỉnh Đăk Lăk

Điện thoại: 0262.3808088 Fax: 0262.3808088

E-mail: ctn-nxbttt@mic.gov.vn

In 500 bản, khổ 16x24 cm, tại Công ty In Hải Nam; Địa chỉ: Số 18 ngách 68/53/9
Quan Hoa, Cầu Giấy, Hà Nội.

Số xác nhận đăng ký xuất bản: 2155-2018/CXBIPH/1-92/TTTT

Số quyết định xuất bản: 201/QĐ-NXB TTTT ngày 25 tháng 6 năm 2018

Ín xong và nộp lưu chiểu quý III năm 2018.

Mã số: HT 07 HM 18

Mã ISBN: 978-604-80-3254-8