**CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT**

**16CNTN**

BÁO CÁO

B-TREE

* Môn : Thực hành cấu trúc dữ liệu và giải thuật
* MSSV : 1612052
* Họ tên : Phạm Minh Chiến
* Lớp : 16CNTN
* Chủ đề : ***B-Tree***

C:\Users\tdqua_000\Dropbox\SS-Slides\DeCuong-CDIO\Template CDIO v4.2\Templates\Hinh anh\LogoTruong.png

Khoa Công nghệ Thông tin

Đại học Khoa học Tự nhiên TP HCM

Tháng 10/2017

**B-TREE**

1. **Giới thiệu về B-Tree**

B-Tree là một loại cây cân bằng tìm kiếm được thiết kế để thao tác tốt với dữ liệu được lưu trữ ở bộ nhớ ngoài. B-Tree cũng gần giống với Red-Black Tree (cây đỏ-đen) nhưng được tốt hơn nhiều ở thao tác truy xuất dữ liệu từ bộ nhớ ngoài. Rất nhiều hệ thống cơ sở dữ liệu sử dụng B-Tree (hoặc các dạng mở rộng của B-Tree) để lưu trữ thông tin.

B-Tree khác với cây nhị phân ở chỗ nào có rất nhiều cây con từ vài cây đến hàng ngàn cây con. Và đặc điểm quan trọng nhất mà khi sử dụng cây B-Tree đó là chiều cao của cây, bởi muốn giảm thiểu thao tác với bộ nhớ ngoài thì chiều cao của cây phải càng thấp.

Mỗi node của B-Tree chứa: n khóa, n+1 con trỏ trỏ đến cây con. N khóa của node được lưu thành dãy các khóa, n+1 con trỏ của node sẽ trỏ đến n+1 dãy con của node đó.

1. **Định nghĩa cấu trúc B-Tree**

Cấu trúc của B-Tree bao gồm các node gọi là “trang”, bao gồm trang trong (internal) và trang lá (leaf). Trang trong chứa các liên kết quản lí trang trong con hoặc trang lá. Trang lá không quản lí trang nào khác.

* Mỗi node **x** của B-Tree có các thuộc tính:
  + *n*: số lượng khóa được lưu trữ.
  + Dãy N khóa được lưu có thứ tự không giảm: x.key0 <= x.key1 <=...<= x.keyn-1.
  + Leaf: biến boolean lưu TRUE nếu là trang lá và ngược lại.
  + Nếu x là trang trong (leaf == FALSE) thì:
    - Chứa n+1 con trỏ x.c0, x.c1,... x.cn trỏ đến trang con.
  + Tất cả các trang lá đều có cùng chiều cao h.
* Gọi **t** (t>=2)là bậc của B-Tree ta có:
  + Mỗi node trừ node gốc phải có tối thiểu t-1 khóa, t trang con.
  + Mỗi node có tối đa 2t-1 khóa, 2t trang con.

*(Bậc của cây có nhiều cách thể hiện, ở đây dùng đinh nghĩa “minimum degree” – số lượng tối thiểu trang con làm bậc của cây).*

* + - * *Khai báo struct B-Tree C++:*

|  |
| --- |
| const int t = 2; // giả sử B-Tree bậc 2  struct Node  {  int key[2t-1]; // tối đa 2t-1 khóa  int n; // số lượng khóa  bool leaf; // biến cờ lá  Node\* c[2t]; // tối đa 2t con trỏ trỏ đến trang con  } |

* + - * Chiều cao cây: cây B-Tree có **m** khóa, với bậc t>=2 thì chiều cao:

1. **Các thao tác cơ bản trên B-Tree**
   1. **Tìm kiếm**

|  |
| --- |
| **Thuật toán tìm kiếm trên B-Tree** |
| B-TREE-SEARCH(Node x, int k)  i = 0  **while** (i<x.n && k>x.key*i*)  i = i + 1  **if** (i< x.n && k == x.keyi)  **return** (x, i)  **else** **if** (x.leaf)  **return** (NIL, -1)  **else**  *DISK-READ(x.ci)*  **return** *B-TREE-SEARCH(x.ci, k)* |

Tìm kiếm trên B-Tree tương tự với trên cây nhị phân, thay vì đi 2 nhánh thì giờ sẽ rẽ theo nhiều nhánh (n +1 nhánh). Đặc biệt ta sẽ tạo một cấu trúc chứa kết quả tìm kiếm (Node y, Index i) để lưu Node chứa khóa cần tìm, và vị trí của khóa đó trong Node y. Ví dụ tìm khóa k ta có *y.keyi = k*, nếu không tồn tại trả về (NIL, -1).

* Hàm DISK-READ(x.c*i*) sẽ đọc Node x.ci từ bộ nhớ ngoài vào bộ nhớ trong.
* Hàm WRITE-READ(Node) sẽ ghi Node x.ci ra bộ nhớ ngoài.
  1. **Khởi tạo một cây rỗng**

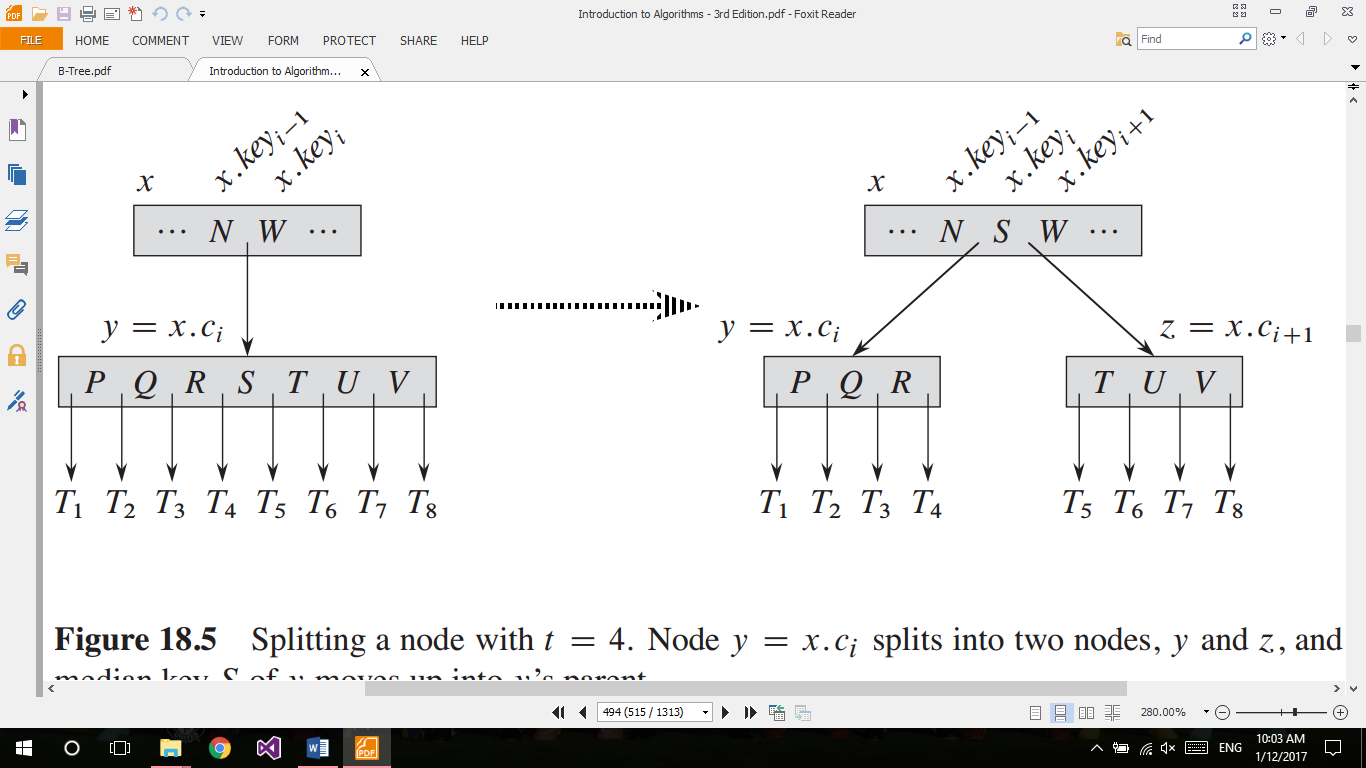
|  |  |
| --- | --- |
| **ALLOCATE-NODE** | **B-TREE-CREATE** |
| ALLOCATE-NODE  **return** *new* Node | B-TREE-CREATE(T)  x = ALLOCATE-NODE()  x.leaf = TRUE  x.n = 0  DISK-WRITE(x)  T = x |

Để xây dựng một cây B-Tree, ta phải sử dụng B-TREE-CREATE để tạo ra một node gốc rỗng sau đó mới chèn bằng B-TREE-INSERT, cả hai đều dùng hàm ALLOCATE-NODE để khởi tạo, cấp phát vùng nhớ cho Node.

* 1. **Chèn một khóa vào cây**

|  |
| --- |
| **Tách node** |
| B-TREE-SPLIT-CHILD(x,i)  z = ALLOCATE-NODE()  y = x.ci  *z.leaf = y.leaf*  z.n = t-1  **for**( j = 0 **to** t-2) //đưa t-1 khóa cuối của y sang z  z.keyj = y.keyj+t  **if**(*y.leaf == false*)  **for**( j = 0 **to** t-1) //đưa t con trỏ cuối của y sang z  z.cj =y.cj+t  y.n = t-1 //y chỉ còn quản t-1 khóa  **for**( j = x.n **downto** i+1) //dời các phần tử sau i qua phải 1  x.cj+1 = x.cj //đv để chỗ chophần tử mới lên  x.ci+1 = z // cho con trỏ thứ i+1 của x trỏ tới z  **for**(j =x.n-1 **downto** i) //dời các khóa của x lên 1 đơn vị  x.keyj+1 = x.keyj  x.keyi = y.keyt //đẩy y.keyt lên x tại vị trí keyi  x.n = x.n + 1 //cập nhật số lượng khóa của x  DISK-WRITE(y)  DISK-WRITE(z)  DISK-WRITE(x) |

Ta không thể chèn giống như cách chèn vào cây nhị phân được vì như thế rất dễ làm cho Node của B-Tree trở thành không hợp lệ. Cho nên ta có cách chèn sau: thêm trực tiếp vào Node lá  **y** (đúng vị trí theo tiêu chuẩn tìm kiếm), trường hợp lá chưa đầy thì thêm bình thường, nếu lá đầy thì phải **tách** ra thành y và z (thực tế chỉ tạo thêm z), và đẩy node trung tâm của y lên Node cha của y. Khi đó cả y và z đều chưa đầy nên thêm vào. Quá trình này lặp lại với cha của y nếu cha y đã đầy trước khi đưa phần tử trung tâm của y lên. Cứ như vậy cho đến khi lên đến gốc thì sẽ tăng chiều cao cây. Tuy nhiên thiết kế hàm B-TREE-SPLIT-CHILD dưới đây là trường hợp cha của **y** không đầy, quá trình chèn sẽ kiểm soát trường hợp đầy. Tách y ra bằng cách gọi SPLIT(y’parent, i).



*(Minh họa tách x,i)*

Như vậy để sử dụng được hàm tách này thì node x cần chưa đầy mới có thể chứa thêm 1 phần tử đưa từ dưới lên. Như vậy ta thiết kế chèn như sau:

|  |  |
| --- | --- |
| **Chèn một khóa vào cây** | **Chèn vào Node chưa đầy** |
| B-TREE-INSERT(T,k)  r=T;  if(r.n == 2t-1) // full  s = ALLOCATE-NODE()  T=s  s.leaf = false  s.n = 0  s.c0 = r  B-TREE-SPLIT(s,0)  B-TREE-INSERT-NONFULL(s,k)  Else  B-TREE-INSERT-NONFULL(r,k) | B-TREE-INSERT-NONFULL(x, k)  i = x.n – 1  **if**(x.leaf == TRUE)  **while**(x>=0 && k<x.keyi)  x.keyi+1 = x.keyi  i=i-1  x.keyi+1 = k  x.n = x.n +1  DISK-WRITE(x)  **else**  **while**(i>=0 && k<x.keyi)  i = i-1  i=i+1  DISK-READ(x.ci)  **if**(x.ci.n == 2t-1)  B-TREE-SPLIT-CHILD(x,i)  **if**(k>x.keyi)  i=i+1  B-TREE-INSERT-NONFULL(x.ci,k) |

* 1. **Xóa một khóa trong cây**

Xoá một khóa trong cây phức tạp chèn một chút vì khi xóa thì có thể khóa nằm ở node trong, node lá, có thể xóa xong làm mất cân bằng hoặc không. Cho nên ta chia thành các trường hợp sau để xóa:

1. Nếu khóa k nằm tại x là node lá: xóa k trong x.
2. Nếu khóa k nằm tại x là node trong:
   1. Tìm phần tử thế mạng k’ cho k, (thuộc node lớn nhất bên con trái **y** hoặc nhỏ nhất bên con phải **z**) với điều kiện k’ có ít nhất t khóa. Sao chép k’ sang k. Xóa k’ (node lá).
   2. Nếu cả y và z đều chỉ có t-1 khóa:

Gộp k và toàn bộ khóa trong z vào y, khi đó x đã mất k và con trỏ để trỏ đến z, hủy z và xóa k trong k là xong.

1. Nếu k không xuất hiện ở node trong x: xác định x.c*i* chứa k. Nếu x.ci chỉ có t-1 khóa thì có hai khả năng:
   1. Node anh em của x.ci (x.ci-1 hoặc x.ci+1) có t khóa: xóa k trong x.ci, đưa khóa x.keyi xuống x.ci, đưa khóa của x.ci-1 hoặc x.ci+1 lên cho x.
   2. Nếu node anh em của x.ci cũng chi có t-1 khóa: xóa k trong x, gộp x.ci với một trong hai x.ci-1 và x.ci+1, đẩy x.key*i*từ x xuống node vừa gộp tại vị trí làm trung gian.
2. **Tài liệu tham khảo**

* Introduction to Algorithm – 3th Eddion – *Thomas H.Cormen, Charles E.Leiserson, Ronald L.Rivest, Cliffrold Stein.*