

Lập Trình C (cơ bản)

Chương 06. Cây nhị phân

Soạn bởi: TS. Nguyễn Bá Ngọc

2021

Nội dung

- Cây nhị phân và cây nhị phân tìm kiếm
- Một số thao tác tiêu biểu
- Các bài tập

Cấu trúc cây nhị phân

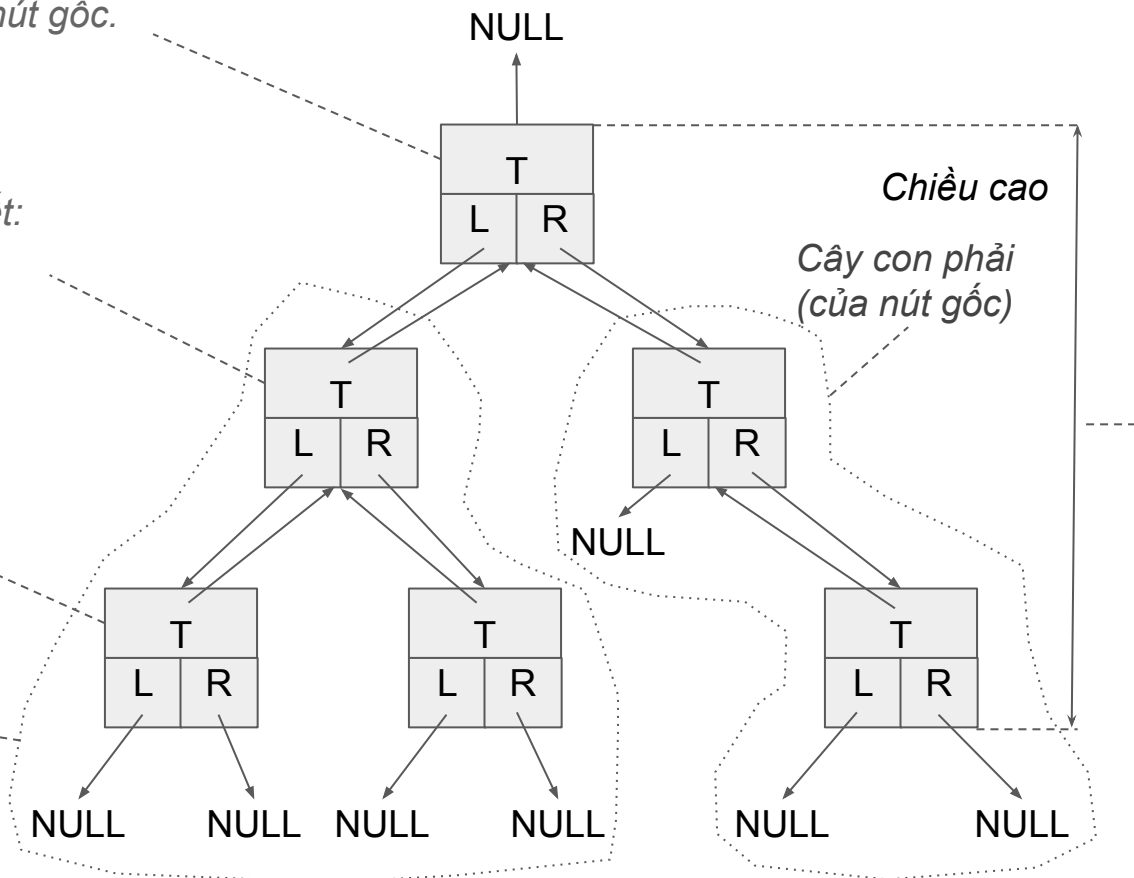
Mỗi cây chỉ có đúng 1 nút gốc.
Nút gốc có $T == \text{NULL}$.

Mỗi nút trong hình vẽ có chứa 3 liên kết:

- L (left), - Con trỏ tới nút con trái;
- R (right) - nút con phải;
- T (top) - nút cha.

Các nút lá (leaf) có $L == \text{NULL}$
&& $R == \text{NULL}$.

Cây con trái
(của nút gốc)



Chiều cao của cây có thể được định nghĩa dựa trên cạnh hoặc dựa trên đỉnh:

- = số lượng cạnh trong đường đi dài nhất từ nút gốc tới nút lá, hoặc
- = số lượng nút trong đường đi dài nhất từ nút gốc tới nút lá.
- Chiều cao tính theo nút = chiều cao tính theo cạnh + 1.

Biểu diễn cây nhị phân trong C

- Biểu diễn nút:

```
typedef struct bn_node {  
    struct bn_node *left;  
    struct bn_node *right;  
    struct bn_node *top;  
} *bn_node_t;
```

Với các con trỏ left và right chúng ta có thể duyệt cây theo hướng từ gốc tới lá, con trỏ top giúp di chuyển dễ dàng hơn giữa các nút trong cây.

- Biểu diễn cây:

```
typedef struct bn_tree {  
    bn_node_t root;  
} *bn_tree_t;
```

Trong thực tế còn có nhiều biểu diễn cây nhị phân khác nữa mà chúng ta không tìm hiểu hết trong học phần này.

Cây nhị phân tìm kiếm

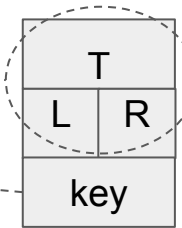
Như có thể thấy từ tên gọi: Cây nhị phân tìm kiếm là một dạng *cây nhị phân* được thiết kế sao cho có thể *tìm kiếm nhanh* một nút bất kỳ trong cây. Chúng ta sẽ tổ chức cây nhị phân tìm kiếm như cây nhị phân với các mở rộng sau:

- Bổ xung vào cấu trúc nút của cây nhị phân một thuộc tính có kiểu sao cho có thể so sánh được các giá trị của nó. Chúng ta gọi thuộc tính này là thuộc tính khóa (key).
- Các nút của cây nhị phân tìm kiếm được bố trí sao cho các giá trị của thuộc tính khóa thỏa mãn *tính chất cây nhị phân tìm kiếm*: Với x là một nút bất kỳ trong cây và y là một nút bất kỳ khác trong cây con của x . Nếu y là nút trong cây con trái của x thì $y.key \leq x.key$, nếu ngược lại (y là nút trong cây con phải của x) thì $y.key \geq x.key$.

Biểu diễn cây nhị phân tìm kiếm trong C

- Biểu diễn nút:

```
typedef struct bns_node_g {  
    struct bn_node base;  
    gtype key;  
} *bns_node_g_t;
```



- Biểu diễn cây:

```
typedef struct bns_tree_g {  
    struct bn_tree base;  
    bn_compare_t cmp;  
} *bns_tree_g_t;
```

Hàm so sánh các khóa

*Trong triển khai của chúng ta: Nếu y là nút trong cây con trái của x thì **y.key < x.key**, nếu ngược lại (y là nút trong cây con phải của x) thì **y.key >= x.key**.*

Kiểu dữ liệu của thuộc tính khóa

- Trong thực tế có những trường hợp cần sử dụng khóa với kiểu số nguyên, nhưng cũng có những trường hợp cần sử dụng khóa với kiểu khác, ví dụ chuỗi ký tự.
 - Triển khai cây nhị phân tìm kiếm trong những trường hợp này có rất nhiều điểm chung.
 - **Làm sao để hạn chế trùng lặp mã nguồn?**
- Một giải pháp tương đối đơn giản để có thể tái sử dụng tối đa 1 triển khai cho nhiều trường hợp với các khóa có kiểu khác nhau và hạn chế trùng lặp mã nguồn là biểu diễn khóa như 1 nhóm kiểu (kiểu gộp, union).

Tương tự như với chuỗi ký tự, với kiểu gộp chúng ta phải sử dụng hàm để so sánh, không sử dụng được các toán tử lô-gic.

Kiểu dữ liệu của thuộc tính khóa₍₂₎

- Chúng ta sẽ sử dụng kiểu gtype cho khóa:

```
typedef union {
```

```
    long i;
```

```
    double d;
```

```
    char *s;
```

```
    void *v;
```

```
} gtype;
```

Có thể tiếp tục mở rộng danh sách với các kiểu dữ liệu khác.

- Hàm so sánh các giá trị gtype được định nghĩa theo tình huống sử dụng. Chúng ta sẽ sử dụng các giá trị trả về tương tự như strcmp.
- Ví dụ, int gcmp(gtype v1, gtype v2);
 - Nếu $v1 < v2$ thì trả về giá trị < 0 ,
 - nếu ngược lại: nếu $v1 > v2$ thì trả về giá trị > 0 ,
 - nếu ngược lại ($v1 == v2$) thì trả về 0.

Ví dụ 6.1. Giới thiệu gtype

```
5  #include "ext/io.h"
6  #include "gtype.h"
7  #include <stdio.h>
8  #include <stdlib.h>
9
10 typedef struct contact {
11     char *number;
12     char *fullname;
13 } *contact_t;
14
15 void print_contact(gtype v) {
16     printf("Chi tiết liên lạc: \n");
17     printf("Số: %s\n", ((contact_t)(v.v))->number);
18     printf("Họ và tên: %s\n", ((contact_t)(v.v))->fullname);
19 }
20
21 int main() {
22     gtype v;
23     printf("Nhập 1 số nguyên: ");
24     scanf("%ld", &v.i);
25     printf("Bạn đã nhập: %ld\n", v.i);
26     printf("Nhập 1 số thực: ");
27     scanf("%lf", &v.d);
28     printf("Bạn đã nhập: %f\n", v.d);
29     contact_t c = calloc(1, sizeof(struct contact));
30     printf("Nhập số: ");
31     clear_stdin();
32     remove_tail_lf(cgetline(&(c->number), 0, stdin));
33     printf("Nhập họ và tên: ");
34     remove_tail_lf(cgetline(&(c->fullname), 0, stdin));
35     v.v = c;
36     print_contact(v);
37     free(c->number);
38     free(c->fullname);
39     free(c);
40     return 0;
41 }
```

Người gọi hàm cần truyền tham số với kiểu và nội dung phù hợp

Ghi và đọc với cùng 1 trường.

Ví dụ 6.2. BNS với kiểu double

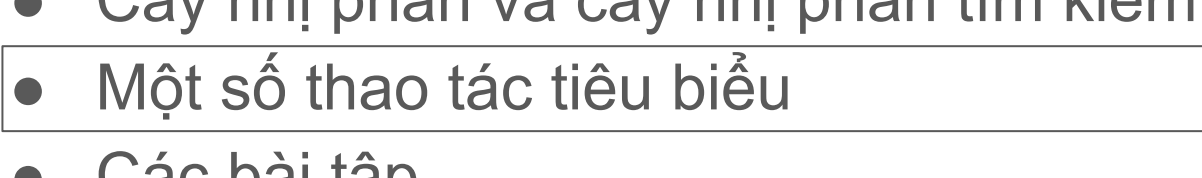
```
9  #include "cgen.ic"
10
11  int main() {
12      bn_tree_t t = bns_create_tree_g(NULL, gtype_cmp_d);
13      double val;
14      printf("Nhập vào một dãy số thực không âm (nhập 1 số âm bất kỳ để kết thúc nhập): \n");
15      for (;;) {
16          scanf("%lf", &val);
17          if (val < 0) {
18              break;
19          }
20          bns_insert_g(t, gtype_d(val));
21      }
22      long sz = bn_size(t);
23      printf("Số lượng số không âm đã nhập: %ld\n", sz);
24      if (sz > 0) {
25          bns_pprint(t, gtype_print_d);
26          printf("Các số theo thứ tự tăng dần: ");
27          bn_traverse_lnr(cur, t) {
28              printf(" %f", bns_node_g_key(cur).d);
29          }
30          printf("\nCác số theo thứ tự giảm dần: ");
31          bn_traverse_rnl(cur, t) {
32              printf(" %f", bns_node_g_key(cur).d);
33          }
34          printf("\n");
35      }
36      bn_free_tree(&t);
37      return 0;
38  }
```

Hàm so sánh các giá trị gtype

Hàm xuất giá trị gtype

Vòng lặp duyệt các nút của cây

Nội dung

- Cây nhị phân và cây nhị phân tìm kiếm
 - Một số thao tác tiêu biểu
 - Các bài tập
- 

Một số thao tác tiêu biểu

- `bn_tree_t bns_create_tree_g(
 bn_node_t root, // Gốc của cây
 bn_compare_t cmp // So sánh các giá trị khóa
);`

Tạo đối tượng cây nhị phân tìm kiếm. Trả về con trỏ tới đối tượng được tạo, hoặc NULL nếu phát sinh lỗi.

- `bn_node_t bns_insert_g(
 bn_tree_t t, // Cây được thêm vào.
 gtype key // Khóa được thêm vào
);`

Thêm key vào cây t. Trả về con trỏ tới nút mới được tạo, hoặc NULL nếu phát sinh lỗi.

Một số thao tác tiêu biểu

- `bn_node_t bns_search_g(
 bn_tree_t t, // Tìm kiếm trong cây t
 gtype key // Khóa cần tìm.
);`

Tìm nút trong cây t có khóa == key. Trả về con trỏ tới nút tìm được, hoặc NULL nếu không tìm thấy.

- `bn_node_t bns_search_gte_g(
 bn_tree_t t, // Tìm kiếm trong cây t
 gtype key // Khóa cần tìm.
);`

Tìm nút trong cây có khóa nhỏ nhất và \geq key. Trả về con trỏ tới nút tìm được, hoặc NULL nếu không tìm thấy.

Một số thao tác tiêu biểu

- `bn_node_t bns_search_lte_g(
 bn_tree_t t, // Tìm kiếm trong cây t
 gtype key // Theo khóa key
);`

Tìm nút có khóa lớn nhất và \leq key. Trả về con trỏ tới nút tìm được, hoặc NULL nếu không tìm thấy.

- `void bns_delete_g(
 bn_tree_t t, // Xóa trong cây t
 bn_node_t n // nút trong cây t cần xóa
);`

Xóa nút n khỏi cây t (điều chỉnh các liên kết) không giải phóng bộ nhớ cho n.

Một số thao tác tiêu biểu₍₂₎

Các vòng lặp duyệt cây:

- `bn_traverse_lnr` - // Duyệt cây theo trình tự trái-giữa-phải
`bn_traverse_lnr(cur, tree) {`
... // Xử lý nút hiện tại (`cur`) trong tiến trình
`}`
- `bn_traverse_rnl` - // phải-giữa-trái
- `bn_traverse_lrn` - // trái-phải-giữa

Tính một số đặc điểm của cây:

- `long bn_size(bn_tree_t t);` - // Số lượng nút trong `t`
- `long bn_edge_height(bn_tree_t t);` // Độ cao của `t`

Có thể tham khảo thêm duyệt cây bằng đệ quy, tuy nhiên duyệt cây theo hình thức vòng lặp được ưa chuộng hơn.

Giải phóng bộ nhớ động


- `void bn_free_tree(bn_tree_t *t);` // Giải phóng bộ nhớ động đã được cấp phát cho cấu trúc liên kết của cây, con trỏ tới cây sau đó được gán = NULL.

Trong trường hợp các nút có chứa con trỏ tới vùng nhớ được cấp phát động thì các vùng nhớ đó phải được giải phóng trước. Ví dụ có thể sử dụng cấu trúc sau:

```
traverse_lrn(cur, t) {  
    giải phóng bộ nhớ được cấp phát động trong nút cur,  
    nhưng giữ nguyên các liên kết.  
}  
bn_free_tree(&t);
```

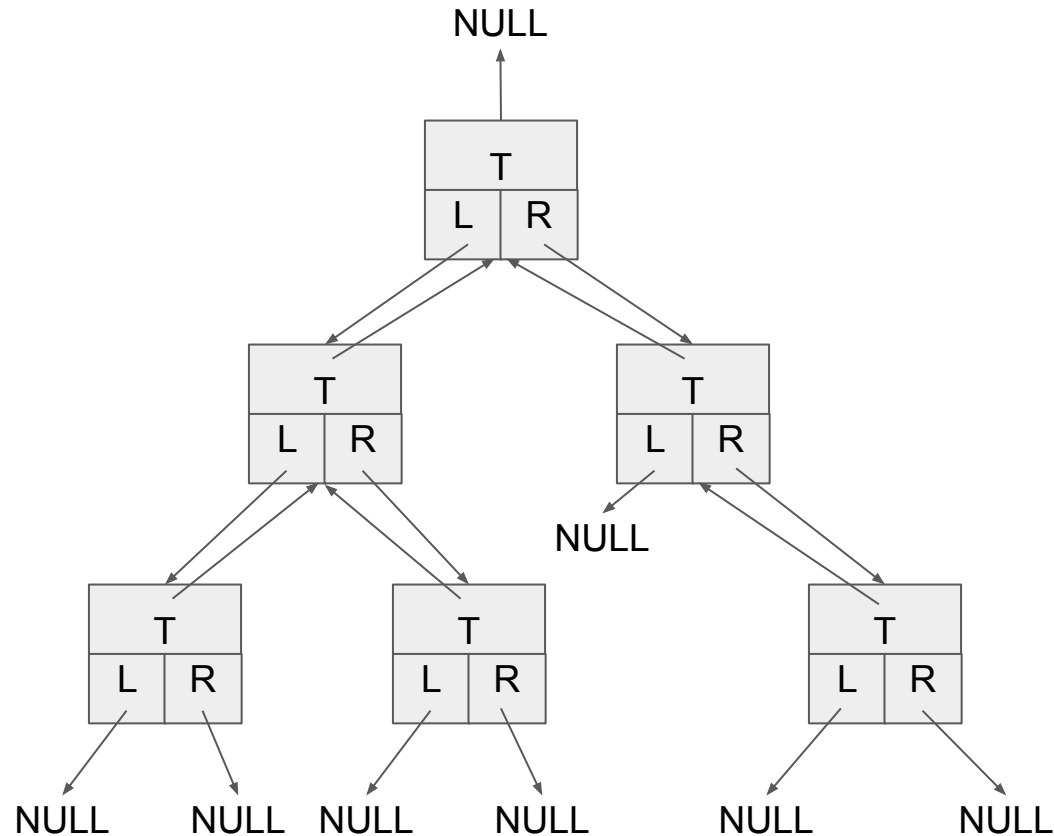
Trong trường hợp ngược lại, thường chỉ cần gọi hàm `bn_free_tree(&t);`

Nội dung

- Cây nhị phân và cây nhị phân tìm kiếm
 - Một số thao tác tiêu biểu
 - Các bài tập
- 

Bài tập 6.1. Cấu trúc liên kết trong cây nhị phân

Viết chương trình tạo cây nhị phân có cấu trúc như trong hình vẽ.
Sau đó chương trình tính số lượng nút, độ cao cây tính theo cạnh và xuất cây với địa chỉ của các nút.



Bài tập 6.2. Tìm nút lá

- Viết hàm nhận 1 tham số đầu vào là một cây nhị phân, hàm trả về một danh sách các nút lá trong cây (dll) theo thứ tự từ trái qua phải. Hàm trả về NULL nếu đầu vào là cây rỗng
`dll_t bn_get_leafs(bn_tree_t t);`
- Phát triển tiếp bài tập 6.1 sử dụng hàm `bn_get_leafs` để đưa ra các nút lá của cây.

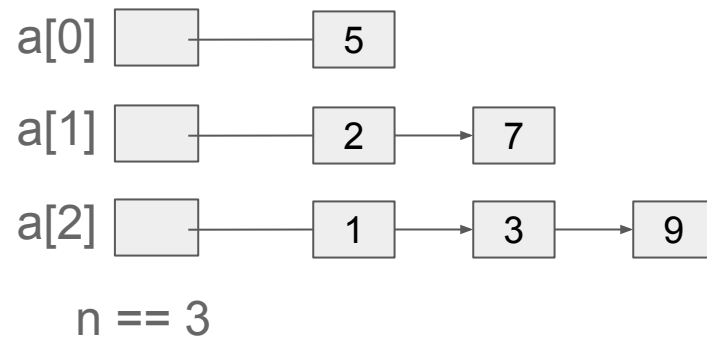
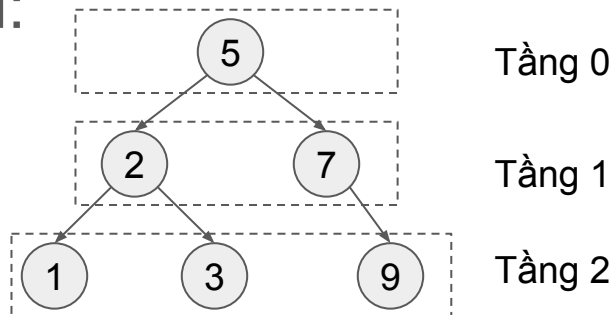
Gợi ý: Duyệt cây (Inr) và kiểm tra từng nút, nếu là nút lá thì đưa vào danh sách.

Bài tập 6.3. Cắt lớp cây

- Viết hàm xử lý cây nhị phân theo tầng (các nút có cùng khoảng cách tới gốc). Hàm trả về một mảng các danh sách 1 chiều (sll), mỗi danh sách chứa các nút của một tầng theo thứ tự từ trái sang phải. Số lượng phần tử trong mảng được trả về qua tham số thứ 2 (long *n).
- Mở rộng bài tập 6.1, sử dụng hàm để xuất các tầng của cây tạo được.

```
sll_t *bn_layers(bn_tree_t t, long *n);
```

- Ví dụ:



```
long n;
sll_t *a = bn_layers(t, &n);
```

Bài tập 6.3. Hướng dẫn

Có thể sử dụng một hàng đợi FIFO - q , và xử lý theo tầng:

Khởi tạo: Thêm $t \rightarrow \text{root}$ vào hàng đợi q .

Tầng 0 chỉ bao gồm nút gốc (*tất cả các nút hiện có trong q*).

Tầng 1 bao gồm các nút con của tầng 0

...

Chúng ta có tầng i bao gồm các nút con của tầng $i - 1$

Bài tập 6.4. Mảng khóa

- Viết hàm đọc các khóa của một cây nhị phân tìm kiếm theo thứ tự tăng dần:
`gtype *bns_keys(bn_tree_t t, long *n);`
t - Tham số đầu vào, là con trỏ cây nhị phân tìm kiếm.
n - Tham số đầu ra - lưu số lượng phần tử mảng.
Hàm trả về con trỏ mảng được cấp phát.
- Có thể điều chỉnh ví dụ 6.2, sử dụng hàm `bns_keys` để xuất các giá trị số trong cây.

Gợi ý: Duyệt cây nhị phân tìm kiếm theo thứ tự Inr.

Bài tập 6.5. Từ điển

- Viết chương trình đưa ra danh sách các từ duy nhất có trong một tệp văn bản. Đường dẫn tới tệp văn bản được truyền cho chương trình qua tham số dòng lệnh đầu tiên.
- Trong phạm vi bài tập này chúng ta sử dụng một định nghĩa thô sơ từ là một chuỗi các ký tự liên tiếp không chứa khoảng trắng.

Gợi ý: Có thể đọc từng từ bằng `fscanf("%s", ...)`. Sử dụng 1 cây nhị phân tìm kiếm để lưu các từ, kiểm tra từ trong cây trước khi thêm vào. Duyệt cây để đưa ra danh sách từ.

Bài tập 6.5. Ví dụ minh họa

- Đặt tên chương trình là `wu`:
- Cho tệp văn bản `story.txt` với nội dung như sau
Hà_Nội mùa_thu mùa_thu Hà_Nội
Mùa_hoa_sữa về thơm từng cơn_gió
- Sau khi chạy chương trình `./wu story.txt`, chương trình cần xuất ra màn hình các thông tin tương tự như sau:

Số lượng từ duy nhất: 7

Các từ duy nhất: Hà_Nội mùa_thu Mùa_hoa_sữa về
thơm từng cơn_gió.

- Có thể xuất các từ theo thứ tự khác với ví dụ.

Bài tập 6.6. Độ cao của cây

- Viết chương trình sinh ngẫu nhiên 1000 000 số nguyên và thêm vào cây nhị phân tìm kiếm theo thứ tự sinh ngẫu nhiên. Quan sát các đặc điểm sau của cây thu được sau mỗi lần thêm vào 10000 số:
 - So sánh với chiều cao thực tế của cây và chiều cao tối ưu (tối thiểu) của cây với cùng số lượng nút.
 - So sánh chiều cao của cây con trái và cây con phải của nút gốc?
- Có thể kết luận gì từ các dữ liệu quan sát được?

Gợi ý: Chiều cao tính theo nút của cây nhị phân đầy đủ với n nút là $1 + (\text{long})\log_2(n)$.

