2017 年 7 月 14 日　米村作成！

binarytree.c は、0 より大きい整数値を入力すると二分探索木を作成し、通りがけ順によって各ノードの値を出力するプログラムです！同じ値が入力されない想定でプログラムを作っているので、使用時はご注意を！

例えば、このプログラムをコンパイルして実行し、10、8、4、15、9、26、35、12、13、21、7、-1 という順番に数値を入力すると（実際の入力では数値の後にエンターキーお願いしまっす）、図 1 に示す二分探索木が作成されますねー！

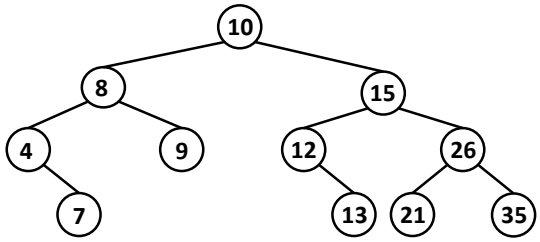


図 1　作成された二分探索木

さて、削除を追加しましょう！main()の前半部分にて、二分探索木を構成する数値を入力するループがあり、0 以下の数値（-1 など）を入力すると作成が終了し、作成された二分探索木が通りがけ順により出力されます！次のループは、削除したい値を入力すると、その値を持つノードが削除されることを想定して作ってあって、入力の度に、削除が行われ削除後の二分探索木が出力されます！ただし、現時点では削除機能を持つ delete()は未完成で（えーっ！）、delete()内にて使用する search()も同様に未完成で（えーっ！）、数値の入力は受け付けるけど、削除されないわけです！

まずは search()を作成しよう！

この関数では、削除したいノードを指すポインタのアドレスが分かると良いので、それを探索する！search()が delete()から呼び出された際の p は NODE 型ポインタ変数 root を指している（図 2 左上部分の「p は最初 root を指す」を参照）！これは main()にて delete()を呼び出す際に、root のアドレスを渡していることからも伺えますね！

例えば、削除したいノードが 58 のノードであった場合には、p は 58 のノードを指すポインタのアドレスを持つようにして（図 2 右上部分の「58 のノードを指すポインタのアドレスを持つように更新されていく」を参照）、それを返してあげれば良いので、そのような考え方で作ってみます！

うまく動けば、delete()内で search()を呼び出している部分にて適切な値が wp に返され、その次の部分で、探している値が出力されると思います！そういうプログラムを予め書いておきましたので！

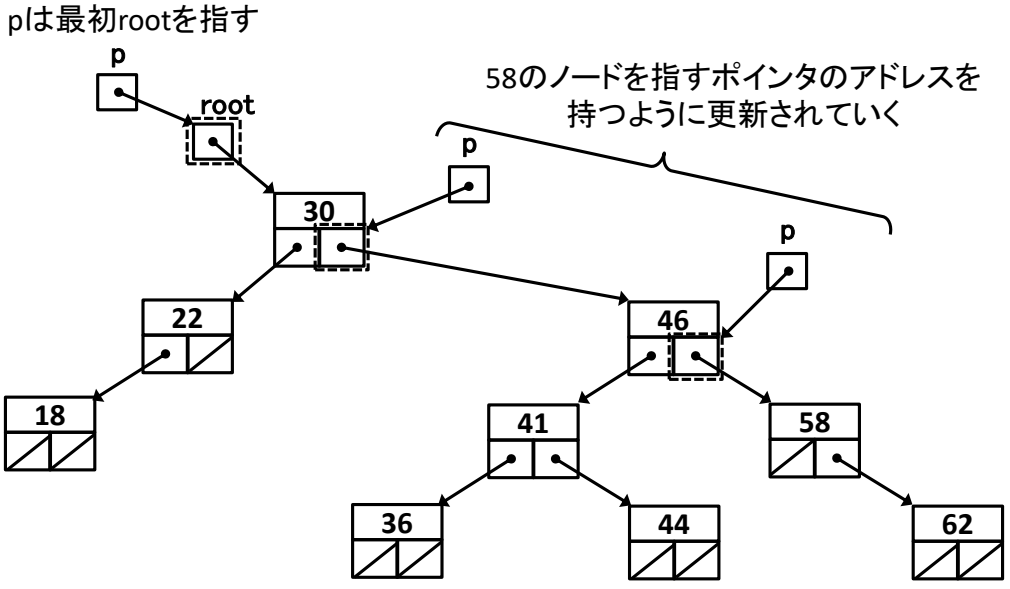


図 2　search()内にて削除したいノードを指すポインタのアドレスを持つように p を更新していく

では、search()のアルゴリズムですが、

■ *p が NULL でない間、以下を繰り返す

もし data が (*p)->value と同じであれば

p を返す （見つかったので）

そうでないときに、もし data が(*p)->value より小さいなら

p を&(*p)->left にて更新する

そうでないなら （具体的には、data が(*p)->value より大きいなら）

p を&(*p)->right にて更新する

■ NULL を返す

となります！

このアルゴリズムを具体例で見ていくと、58 を探している際には（data が 58)、図 3 の一番左の p の様子から始まって、*p(つまり root ですね)は 30 のノードを指している(NULL ではない)ので、data(は 58)と 30(これは(*p)->value ですね)を比較して、58 は 30 より大きいですから、p は「p が指しているポインタが指しているノードのメンバ変数 right のアドレス（プログラム的には&(*p)->right)」で更新されます！そうすると図 3 の真ん中の p の状態になりますね！まだ、*p は NULL ではありませんから（46 のノードを指している)、data（は 58）と 46 を比較します！その結果、同じように p が更新され、図 3 の一番右の p の状態になりますね！の状態にて、data（は 58）と 58（(*p)->value）が比較されて、一致するので p を返すわけです！elete()の wp がこの p を受け取るので、(*wp)->value は 58 となりますね。それを printf()にて出力しています！

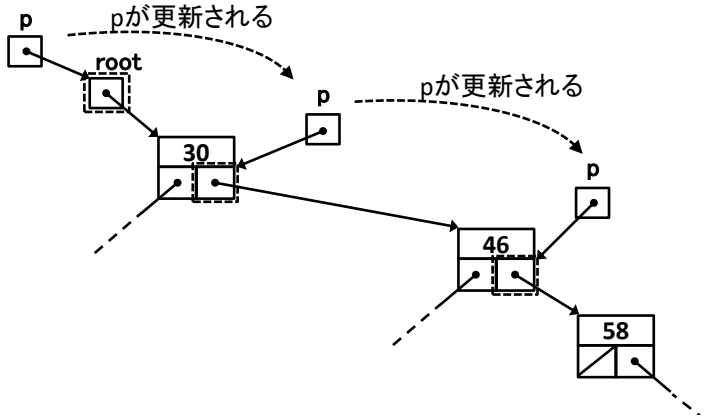
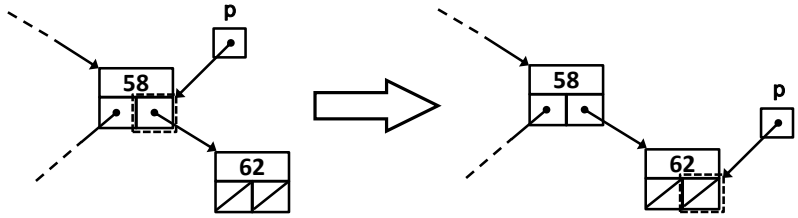


図 3 search()の中で p が更新されていく様子

ところで、p を更新していった結果、葉の left や right にて更新されると、*p == NULL となります（図 4 右側の p を参照)！こうなった時にはループを抜けます！そして、これは data が見つからなかったことを意味していますので、NULL を返し、呼び出し側の delete()の wp は NULL を受け取ることになります！



pを更新していった結果、*p==NULLとなった例

図 4 data が見つからなかった時のパターンであり、ループを抜けて NULL を返す

さて、search()が上手く動いたら、いよいよ削除ですね！削除の方法では、次の 3 つの場合を考えます！

①削除したいノードが葉の場合

②削除したいノードのどちらか片方に子がある場合

③削除したいノードの両方に子がある場合

ということで、まずは①に対応しましょう！delete()の中で、この 3 つの場合分けを行っている部分を見てください！最初の if 文の条件(*wp)->left == NULL && (*wp)->right == NULL は、今見ているノードが葉だったらという意味ですので、例えば、図 5 の左側のような状態ですね！ですので、wp が指しているポインタ型変数に NULL を代入すれば、削除したことになりますね！プログラム的には、*wp に NULL を入れると良いです！やってみましょう！

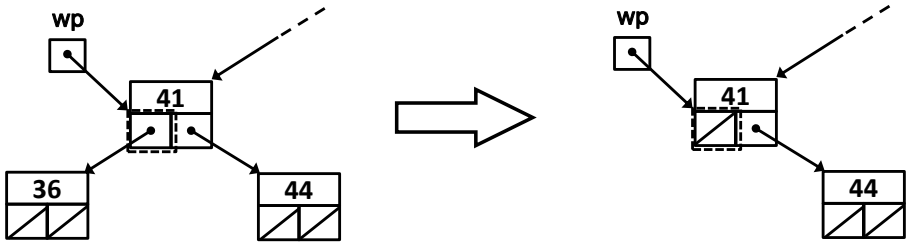


図 5　wp が指しているポインタ型変数に NULL を代入することで削除を実現

次は、②ですね！次の if 文の条件(*wp)->left == NULL || (*wp)->right == NULL は、今見ているノードが（「葉ではなく」は上の if 文で通過済み）left、right のどちらかに子を持つなら、という意味ですので、その場合さらに if 文で right に子を持つ時（(*wp)->left == NULL）と left に子を持つ時とで削除の方法を変えています！

具体的には、46 のノードを削除する際にそのノードが right にのみ子（何らかの部分木）を持つ場合（図 6 左上部分の状態）、その子を指している情報（(*wp)->right ですね）を*wp に代入することで、46 のノードが削除され、*wp が、その子以下の何らかの部分木を指すようになりますね（図 6 下部の状態）！これはこの部分木がどんな形でも構わないわけです！とにかく、その何らかの部分木を指せば、部分木の形にかかわらず、削除が実現できます！

left にのみ子を持つ場合（図 6 の右上部分の状態）も同様の手順にて、結果として 46 のノードが削除されます（図 6 下部の状態になりますね）！さぁ、書いてみましょう！

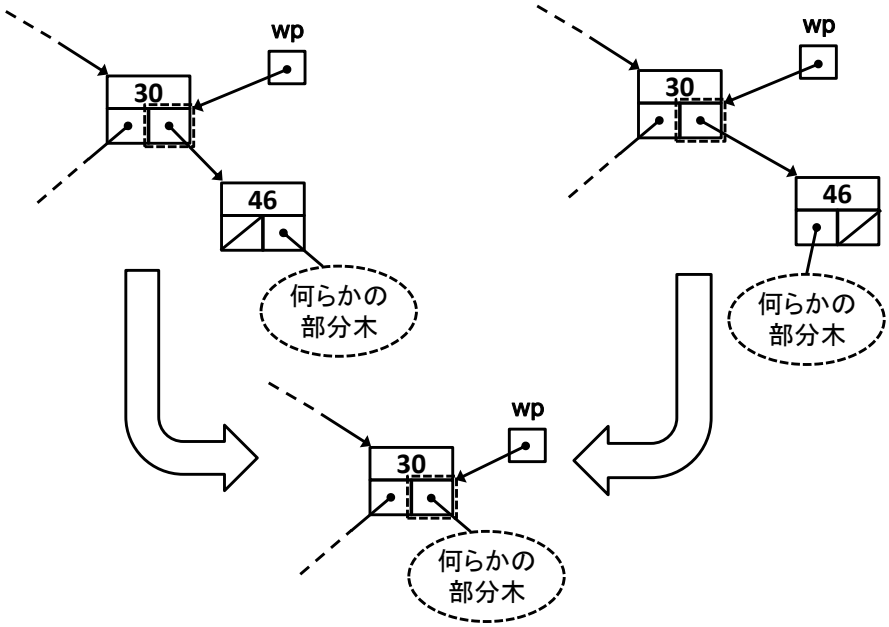


図 6　削除したいノードがどちらか片方のみに子を持つ際の削除の前（上部左右の 2 つ）後（下部）での状態の変化

いよいよ③ですね！図 7 を見てください！削除したいノード 46 が left、right のどちらにも子を持つ場合ですね！この 46 が削除された後に、どのノードがそこへ挿入されれば二分探索木を満たし続けることができるか？を考えると、そうです、この場合、図 8 のようになれば良いわけですね！つまり、削除前の段階での（図 7 での）削除したいノード 46 の左部分木（構成ノードは 41、36、44）の中の最大値である 44 が入ることにより二分探索木の条件が満たされます！

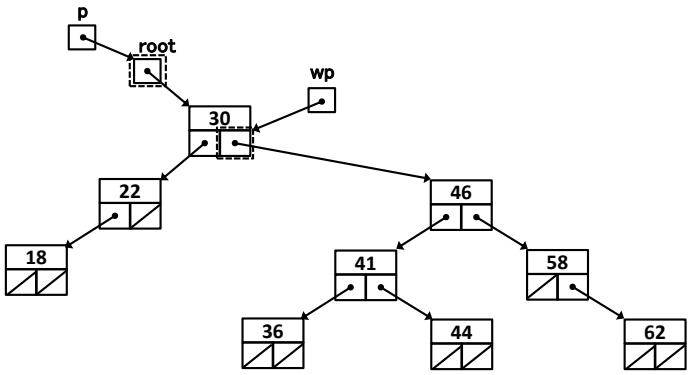


図 7　これから 46 のノードを消すぞって時の、p や wp との関係

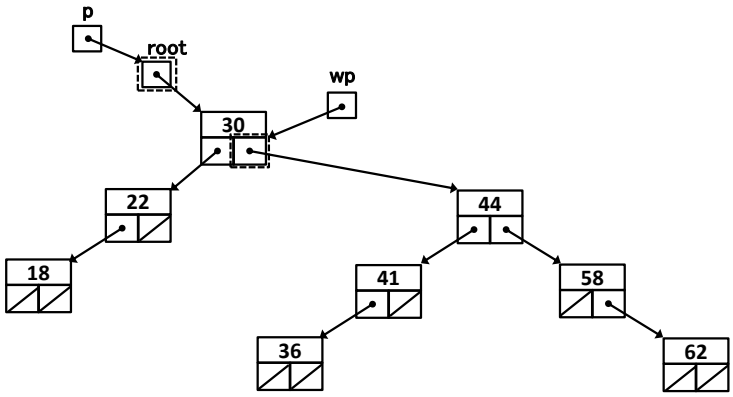


図 8　図 7 から 46 のノードを削除した状態。44 のノードが元あった場所からなくなり、
46 があった場所へ挿入されているのが分かる！

ということで、次は、削除したいノードの左部分木から最大値を持つノードを探す関数 search_max()を作りましょう！
delete()内のプログラム

```
max = search_max(&(*wp)->left);
```

では、search_max()に削除したいノードの left のアドレスを引数として渡していますので（図 9 参照）、これを手掛かりに最大値を（持つノードを）探索することができます！

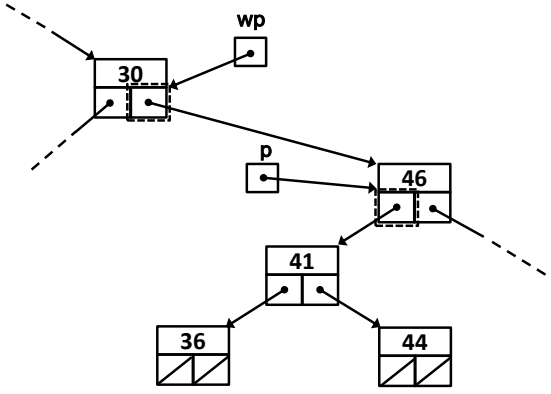


図 9

最大値はどんな条件で探せばよいのでしょうか？二分探索木の性質を考慮すると、right 側の子へ進み、さらに right 側の子へ進みというように、ひたすら right 側の子へ進むことを繰り返していき、right 側の子を持たないノードまで進んだらそれが最大値ですね！最大値のノードが left 側に子を持っていても、調べる必要はありません！left 側の値は、そのノードよりも小さいからです！同様に、調べている途中でも left 側を見る必要はありません！というわけで、プログラムは非常にシンプルなものとなり、

```
(*p)->right  が NULL じゃない間
        p を&(*p)->right へ更新する
```

となりますね！このループを抜けた時、p は最大値を指すポインタ right のアドレスを持っており、(*p)->right は NULL ですね（図 10）！そのアドレスを返して、呼び出し側の delete()の変数 max が受け取ります！

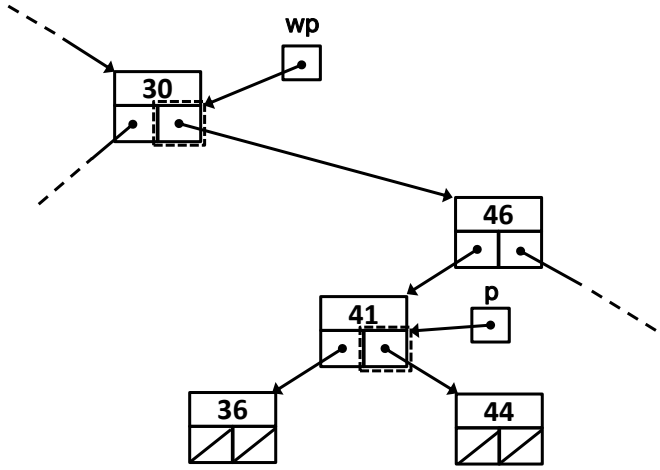


図 10　p は削除したいノードの左部分木の中での最大値を持つノードを指すポインタのアドレスを持つ

これで、削除の準備が整いました！図 11 では、tmp が最大値のノードを指しています（tmp は NODE 型を指すポインタ型であり、ポインタのポインタである必要はありません）！tmp->value を出力すると、search_max()がきちんと機能しているかを確認（削除したいノードの左部分木の中での最大値がでれば正解！）することができますので、そのように binarytree.c に書いておきました！

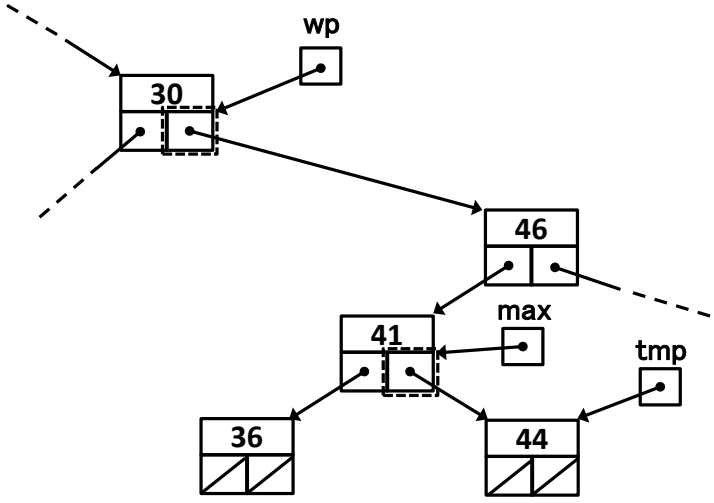


図 11　削除の準備が整った際の、wp、max、tmp の様子

あとは、色々つなぎ替えですね。必要な手順は次のようになるでしょう（これ以降の図は自分で考えてみてください）！

・max が指すポインタ変数の書き換え

- もし、最大値を持つノードが葉であれば、NULL を入れるだけですね！これは①の削除と同じ考え方です！
- そうでなかったら（葉でない）、left が子を持っているのでしょう！②の考え方で削除できますね！
- right が子を持っていることはあり得ません！もし right が子を持っていたら、そちらの値はもっと大きいはずなので、このノードは最大値ではないことになりますから！
- というわけで、max が指すポインタが指しているノードは、葉か left に子があるのみのどちらかの状態なので、①、②のどちらかの方法で削除できるというわけです！

・削除したいノード（ここでは 46）を指しているポインタを指している wp を使ってつなぎ替え

- 削除したいノードの left が持つ情報を tmp->left に代入
- 削除したいノードの right が持つ情報を tmp->right に代入
- これで、削除したいノードが持っている情報が tmp へコピーされましたので、削除できるわけですね！
- 最後に、wp が指しているポインタが、tmp が指しているノードを指すように、代入

これで完成！だと思います！色々入力して確認してみましょう！

さぁ！楽しみましょうー！おー！

以上です！