Lab 9: Recursion, Binary Search Trees, & Linked Lists — Solution

Aim

This lab class comprises pen-and-paper tasks. It will provide you with an opportunity to:

- explore algorithmic recursion — which is important in the processing of self-referential data structures;
- practise manipulating binary search trees; and
- write code to continue to develop your linked-list programming skills.

Tasks

1. Rabbits are very prolific breeders. If rabbits did not die, their population would very quickly get out of hand. The following are some statistics obtained in a recent survey of randomly selected rabbits:
    - rabbits never die;
    - a rabbit reaches sexual maturity exactly two months after birth (i.e. at the beginning of its third month of life); and
    - rabbits are always born in male-female pairs. At the beginning of every month, each sexually mature male-female pair gives birth to exactly one male-female pair.

    (i) Suppose (at the beginning of month 1) that we started with one newborn male-female pair. Using pen and paper, how many pairs would there be in the sixth month, counting the births that took place at the beginning of month 6?

Month 1: 1 pair: the original sexually immature pair
Month 2: ditto
Month 3: 2 pairs: the parent pair and their first offspring
Month 4: 3 pairs: the parent pair, their first offspring and their second offspring
Month 5: 5 pairs: the parent pair, their first, second, and third offspring, and the first offspring of their first offspring
Month 6: 8 pairs: the parent pair, their first, second, third, and fourth offspring, the first and second offspring of their first offspring, and the first offspring of their second offspring.

    (ii) Construct a recursive expression that can be used for computing the value of `Rabbit(n)`, which should be the number of pairs alive in month `n` in terms of the population in early months. *Hint*: start by considering how the number of rabbits alive in the requested month is related to the number alive in the previous month.

If not for sexual maturity,

```
Rabbit(n) = 2 * Rabbit(n-1)
```

However, only those rabbits alive in month n-2 produce offspring in month n, therefore we have:

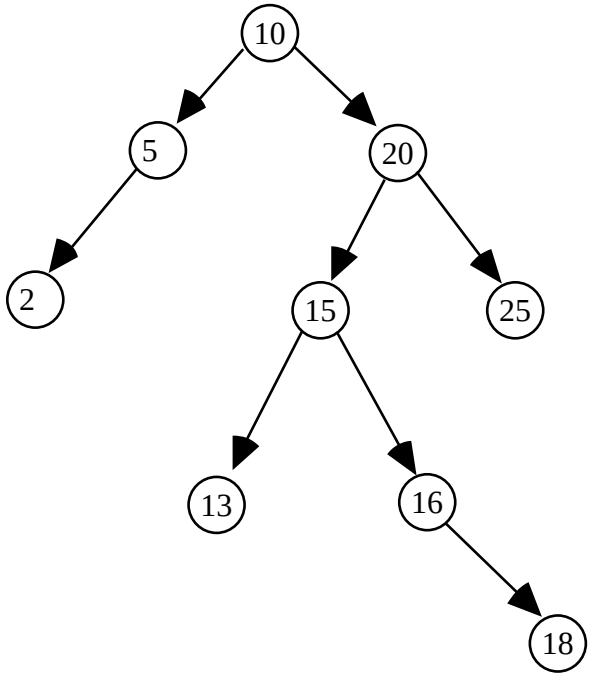
`Rabbit(n) = Rabbit(n-1) + Rabbit(n-2)`

(iii) Consider the terminating case(s) for this scenario. Is there one terminating case for this recursive definition? If so what is it? If not, how many are there and what are they? *Note*: time starts at month 1.

Intuitively we might think that there is one terminating case: month 1 where `Rabbit(1)=1`. *However, in the second month* `Rabbit(2) = Rabbit(1) + Rabbit(0)` *and* `Rabbit(0)` *is earlier than recorded history. Thus there must be two terminating cases:*

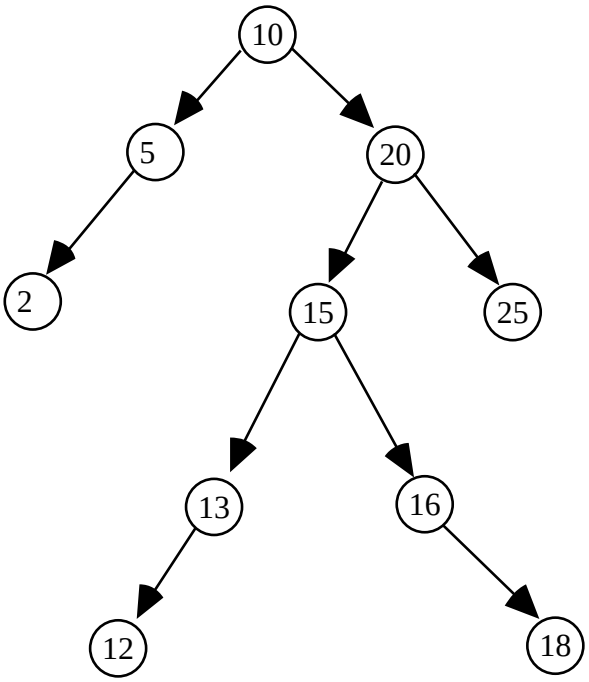
`Rabbit(1)=1` *and* `Rabbit(2)=1`

2. Consider the following binary search tree:



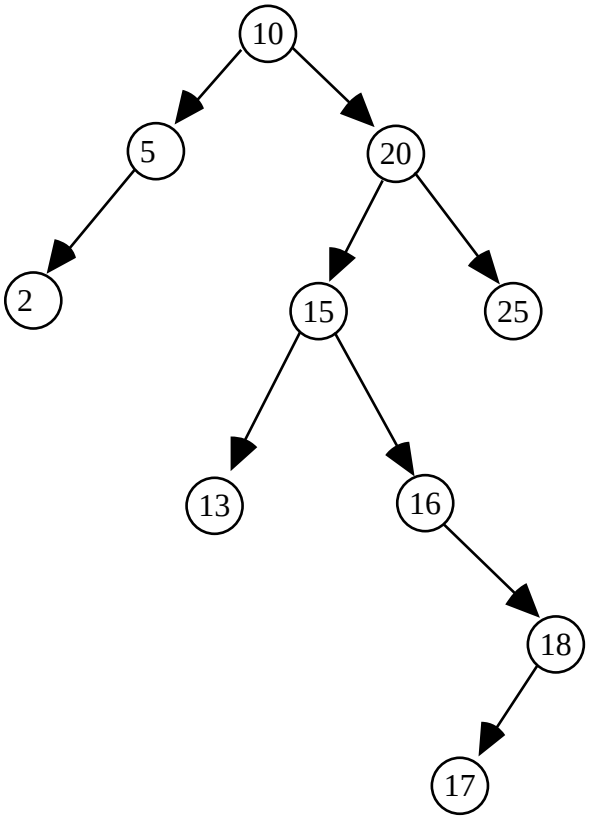
(i) Add value '12' to the tree.

This occurs by tracing a path looking for node 12. When we get to node 13, node 12 should be its left child — it isn't there so we add it:



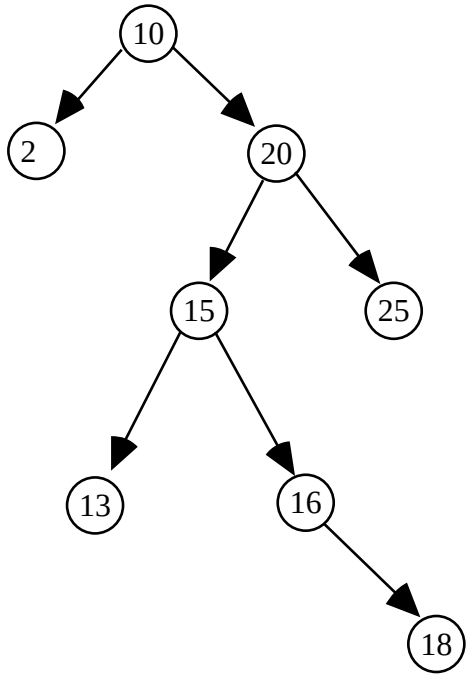
(ii) Add value ‘17’ to the original tree.

Again, tracing a path from the root we look for node 17. It should exist as the right child of node 16, but 18 is there. 17 is less than 18 and so we add a left child:



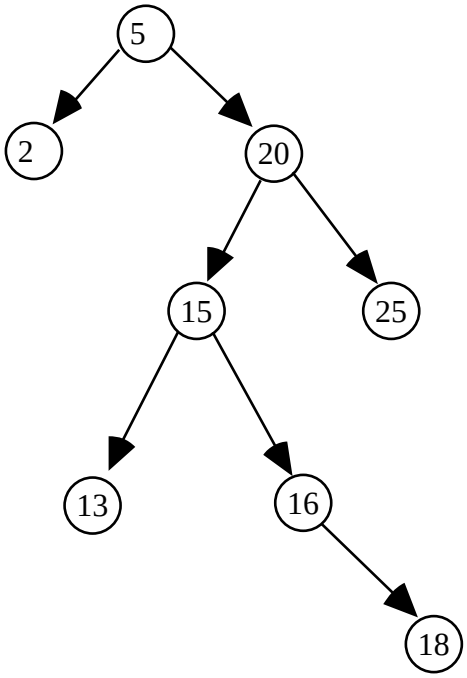
(iii) Delete value ‘5’ from the original tree.

This is a simple matter of dropping a node and replacing it with its child. This produces:



(iv) Delete value '10' from the original tree.

Here we have the deletion of a node with two children. We can choose either the largest node from the smaller subtree or the smallest node from the largest subtree to replace the node and restructure the tree to ensure correct ordering. I have arbitrarily selected node 5:



3. A linked list of integers may be implemented in C using the following types:

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct node_int {
     int data;
     node next;
} *node;

void init_node(node *n, int o)
{
     *n = (node)malloc(sizeof(struct node_int));
     (*n)->data = o;
     (*n)->next = NULL;
}

int get_data(node n)
{
     return (n->data);
}

node get_next(node n)
{
     return (n->next);
}

void set_data(node n, int o)
{
     n->data = o;
}

void set_next(node v, node n)
{
     v->next = n;
}


typedef struct list_int {
     node front;
} *list;
```

Write a C function to delete the largest element in a list — you may assume that the list is not empty and that the largest element value occurs only once. Can this be done with a single traversal of the list?

Naïvely:

```
void remove_max(list t)
{
    node b,c;
```

```
    int m;

    m=get_data(t->front);
    c=get_next(t->front);
    while (c != NULL)
    {
        if ((get_data(c)) > m)
        {
           m=get_data(c);
        }
        c=get_next(c);
    }

    c=t->front;
    b=NULL;
    while ((get_data(c)) != m)
    {
        b=c;
        c=get_next(c);
    }
    if (b == NULL)
    {
        t->front=get_next(c);
    }
    else
    {
        set_next(b,get_next(c));
    }
}
```

Clearly this traverses the list more than once.

An improved version remembers where the maximum occurs, not only what it is:

```
void remove_max(list t)
{
    node b,c,mc,mb;
    int m;

    m=get_data(t->front);
    mc=t->front;
    c=get_next(t->front);
    mb=NULL;
    b=t->front;
    while (c != NULL)
    {
        if ((get_data(c)) > m)
        {
            m=get_data(c);
            mc=c;
            mb=b;
        }
```

```
        b=c;
        c=get_next(c);
    }

    if (mb == NULL)
    {
        t->front=get_next(mc);
    }
    else
    {
        set_next(mb,get_next(mc));
    }
}
```

This solution traverses the list precisely once.