Formal DSA in C++

Nathan Warner

Computer Science
Northern Illinois University
United States

Contents

Elementary complexity theory

1.1 Elementary complexity theory

- **Idea**: The same problem can frequently be solved with algorithms that differ in efficiency. The differences between the algorithms may be immaterial for processing a small number of data items, but these differences grow with the amount of data. To compare the efficiency of algorithms, a measure of the degree of difficulty of an algorithm called computational complexity was developed by Juris Hartmanis and Richard E. Stearns

  Computational complexity indicates how much effort is needed to apply an algorithm or how costly it is. This cost can be measured in a variety of ways, and the particular context determines its meaning. This book concerns itself with the two efficiency criteria: time and space. The factor of time is usually more important than that of space, so efficiency considerations usually focus on the amount of time elapsed when processing data. However, the most inefficient algorithm run on a Cray computer can execute much faster than the most efficient algorithm run on a PC, so run time is always system-dependent. For example, to compare 100 algorithms, all of them would have to be run on the same machine. Furthermore, the results of run-time tests depend on the language in which a given algorithm is written, even if the tests are performed on the same machine. If programs are compiled, they execute much faster than when they are interpreted. A program written in C or Ada may be 20 times faster than the same program encoded in BASIC or LISP.

- **Units**: To evaluate an algorithm's efficiency, real-time units such as microseconds and nanoseconds should not be used. Rather, logical units that express a relationship between the size n of a file or an array and the amount of time t required to process the data should be used

  If there is a linear relationship between the size n and time t, that is, $t_1 = cn_1$, then an increase of data by a factor of 5 results in the increase of the execution time by the same factor. If $n_2 = 5n_1$, then $t_2 = 5t_1$

  Similarly, if $t_1 = \log_2 n$, then doubling n increases t by only one unit of time. Therefore, if $t_2 = \log_2(2n)$, then $t_2 = t_1 + 1$.

- **Eliminating insignificant terms**: A function expressing the relationship between n and t is usually much more complex, and calculating such a function is important only in regard to large bodies of data; any terms that do not substantially change the function's magnitude should be eliminated from the function. The resulting function gives only an approximate measure of efficiency of the original function. However, this approximation is sufficiently close to the original, especially for a function that processes large quantities of data.

- **Asymptotic complexity**: This measure of efficiency is called asymptotic complexity and is used when disregarding certain terms of a function to express the efficiency of an algorithm or when calculating a function is difficult or impossible and only approximations can be found

- **Big-O Notation**: The most commonly used notation for specifying asymptotic complexity—that is, for estimating the rate of function growth—is the big-O notation introduced in 1894 by Paul Bachmann.

  Given two positive-valued functions f and g, consider the following definition:

$f(n)$ is $O(g(n))$ if there exist positive numbers c and N such that $f(n) \leqslant c \cdot g(n)$ for all $n \geqslant N$.

$$f(n) \text{ is } O(g(n)) \iff \exists\, c, N \in \mathbb{Z}^+ \mid f(n) \leq cg(n)\ \forall\ n \geq N.$$

Big-O notation says that for large enough n, the function $f(n)$ does not grow faster than a constant multiple of $g(n)$. So, $g(n)$ provides an upper bound on how fast $f(n)$ can grow as n increases.

In other words, f is big-O of g if there is a positive number c such that f is not larger than $c \cdot g$ for sufficiently large ns; that is, for all ns larger than some number N. The relationship between f and g can be expressed by stating either that $g(n)$ is an upper bound on the value of $f(n)$ or that, in the long run, f grows at most as fast as g.

The problem with this definition is that, first, it states only that there must exist certain c and N, but it does not give any hint of how to calculate these constants. Second, it does not put any restrictions on these values and gives little guidance in situations when there are many candidates. In fact, there are usually infinitely many pairs of c's and N's that can be given for the same pair of functions f and g.

For example, suppose

$$f(n) = 2n^2 + 3n + 1 = O(n^2).$$

Where $g(n) = n^2$. Candidate values for c and N are

c	$\geqslant 6$	$\geqslant 3\frac{3}{4}$	$\geqslant 3\frac{1}{9}$	$\geqslant 2\frac{13}{16}$	$\geqslant 2\frac{16}{25}$	$\cdots$	$\rightarrow$	2
N	1	2	3	4	5	$\cdots$	$\rightarrow$	∞

We obtain these values by solving the inequality:

$$2n^2 + 3n + 1 \leqslant cn^2.$$

Or equivalently

$$2 + \frac{3}{n} + \frac{1}{n^2} \leqslant c.$$

For different n's

For large n, the terms $\frac{3}{n}$ and $\frac{1}{n^2}$ get smaller. Let's find N such that for all $n \geqslant N$, the right-hand side stays bounded.

As n gets larger, $\frac{3}{n}$ and $\frac{1}{n^2}$ approach zero. To simplify the analysis, choose $N = 1$ initially and check how small $\frac{3}{n}$ and $\frac{1}{n^2}$ are:

$$2 + \frac{3}{1} + \frac{1}{1^2} = 2 + 3 + 1 = 6.$$

From the inequality, at $N = 1$, we have $6 \leqslant c$. Therefore, we can choose $c = 6$. This ensures that for all $n \geqslant 1$, the inequality holds:

$$2 + \frac{3}{n} + \frac{1}{n^2} \leqslant 6.$$

Thus, you can choose $c = 6$ and $N = 1$.

different pairs of constants c and N for the same function $g(= n^2)$ can be determined.

- **Choosing the best c, N**: To choose the best c and N, it should be determined for which N a certain term in f becomes the largest and stays the largest.

  In the example above, The only candidates for the largest term are $2n^2$ and $3n$; these terms can be compared using the inequality $2n^2 > 3n$ that holds for $n > 1.5$. Thus, $N = 2$ and $c \geqslant \frac{15}{4} = 3.75$.

- **Significance**: What is the practical significance of the pairs of constants just listed? All of them are related to the same function $g(n) = n^2$ and to the same $f(n)$. For a fixed g, an infinite number of pairs of c's and N's can be identified. The point is that f and g grow at the same rate. The definition states, however, that g is almost always greater than or equal to f if it is multiplied by a constant c. "Almost always" means for all n's not less than a constant N. The crux of the matter is that the value of c depends on which N is chosen, and vice versa.

- **Inherent imprecision: Choosing best $g(n)$**: The inherent imprecision of the big-O notation goes even further, because there can be infinitely many functions g for a given function f. For example, the f from Equation 2.2 is big-O not only of n^2, but also of n^3, n^4, ..., n^k, ...for any $k \geqslant 2$. To avoid this embarrassment of riches, the smallest function g is chosen, n^2 in this case.

- **Big-o as approximating terms**: The approximation of function f can be refined using big-O notation only for the part of the equation suppressing irrelevant information. For example, in the equation below, the contribution of the third and last terms to the value of the function can be omitted

  $$\begin{aligned} & f(n) = n^2 + 100n + \log(n) + 1000 \\ \implies & f(n) = n^2 + 100n + O(\log(n)). \end{aligned}$$

  Similarly,

  $$\begin{aligned} & f(n) = 2n^2 + 3n + 1 \\ \implies & f(n) = 2n^2 + O(n). \end{aligned}$$

  This equation says that for large values of n, the expression $2n^2 + 3n + 1$ behaves like $2n^2$ plus some terms that grow linearly or slower (captured by $O(n)$). The exact contributions of $3n$ and 1 are not important for asymptotic analysis; what matters is that their growth is slower compared to $2n^2$.

- **Algorithm analysis: Most common time complexities**: Ranked slowest to fastest growth

  - $O(1)$**:** Constant time
  - $O(\log(\log(n)))$: Logarithmic time
  - $O(\log(n))$: Logarthmic time
  - $O(n)$: Linear time
  - $O(n \log(n))$: Log-linear time
  - $O(n^k),\ k > 1$: Polynomial time
  - $O(a^n),\ a > 1$: Exponential time
  - $O(n!)$: Factorial time

- **Ranking complexities from slowest to fastest: Process**: Given

  (a) $O(25)$
  (b) $O(n^{\frac{1}{2}} + \log^2(n))$
  (c) $O(\log^{200}(n))$
  (d) $O(n^3 \log^4(n))$
  (e) $O(n^{200} + 3^n)$
  (f) $O(n \log^{40}(n))$
  (g) $O(4^n \log(n))$
  (h) $O(n^3 \log(\log(n)))$

  How can we go about sorting these slowest to fastest. Well, to start, in the expressions with plus or minus, we can throw out the slower terms. Thus,

  (a) $O(n^{\frac{1}{2}})$
  (b) $O(25)$
  (c) $O(\log^{200}(n))$
  (d) $O(n^3 \log^4(n))$
  (e) $O(3^n)$
  (f) $O(n \log^{40}(n))$
  (g) $O(4^n \log(n))$
  (h) $O(n^3 \log(\log(n)))$

  In product terms, we disregard the slower term unless there are complexites with the same dominant term. For example, $O(n^3 \log(\log(n)))$ grows slower than $O(n^3 \log^4(n))$ because although they have the same dominant term n^3, $\log(\log(n))$ grows slower than $\log^4(n)$. Thus, the correct sequence is

  (b) $O(25)$
  (c) $O(\log^{200}(n))$
  (a) $O(n^{\frac{1}{2}} + \log^2(n))$
  (f) $O(n \log^{40}(n))$
  (h) $O(n^3 \log(\log(n)))$
  (d) $O(n^3 \log^4(n))$
  (e) $O(n^{200} + 3^n)$
  (g) $O(4^n \log(n))$

- **Properties of Big-O notation**

  1. **Transitivity**: If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.

     Proof: According to the definition, $f(n)$ is $O(g(n))$ if there exist positive numbers c_1 and N_1 such that $f(n) \leqslant c_1 g(n)$ for all $n \geqslant N_1$, and $g(n)$ is $O(h(n))$ if there exist positive numbers c_2 and N_2 such that $g(n) \leqslant c_2 h(n)$ for all $n \geqslant N_2$. Hence, $c_1 g(n) \leqslant c_1 c_2 h(n)$ for $n \geqslant N$ where N is the larger of N_1 and N_2. If we take $c = c_1 c_2$, then $f(n) \leqslant ch(n)$ for $n \geqslant N$, which means that f is $O(h(n))$.

  2. **Addition**: If $f(n)$ is $O(h(n))$ and $g(n)$ is $O(h(n))$, then $f(n) + g(n)$ is $O(h(n))$.

Proof: If $f(n) \leqslant c_1h(n)$, and $g(n) \leqslant c_2h(n)$, then $f(n) + g(n) \leqslant c_1h(n) + c_2h(n) \leqslant (c_1+c_2)h(n)$. ℓet $c = c_1 + c_2$, then $f(n)+g(n) \leqslant ch(n)$ and $f(n)+g(n)$ is $O(h(n))$

3. **Polynomial bounds**: The function an^k is $O(n^k)$

   Proof: $an^k \leqslant cn^k$ for $c \geqslant a$. Since we can always find some constant $c \geqslant a$, an^k is $O(n^k)$

   Observation: For $an^k \leqslant cn^k$ to hold, $c \geqslant a$ is necessary

4. **Domination of higher-degree polynomials**: n^k is $O(n^{k+j})\ \forall\ j > 0$

   This statement holds if $c = N = 1$

   It follows from all these facts that every polynomial is big-O of n raised to the largest power, or

$$f(n) = a_kn^k + a_{k-1}n^{k-1} + \cdots + a_1n + a_0 \text{ is } O(n^k).$$

5. **Logs**: The function $\log_a n$ is $O(\log_b n)$ for any positive numbers a and $b \neq 1$.

   This correspondence holds between logarithmic functions. The fact above states that regardless of their bases, logarithmic functions are big-O of each other; that is, all these functions have the same rate of growth.

   Proof: Let $\log_a(n) = x$, and $\log_b(n) = y$, then $a^x = n,\ b^y = n.$ Take the natural log of both sides

$$\begin{aligned} \ln(a^x) &= \ln(n) \quad \ln(b^y) = \ln(n) \\ \implies x\ln(a) &= \ln(n) \quad y\ln(b) = \ln(n) \\ \implies x\ln(a) &= y\ln(b). \end{aligned}$$

   Since $x = \log_a(n)$, and $y = \log_b(n)$, then we have

$$\begin{aligned} \ln(a)\log_a(n) &= \ln(b)\log_b n \\ \log_a(n) &= \frac{\ln(b)}{\ln(a)}\log_b(n). \end{aligned}$$

   ℓet $c = \frac{\ln(b)}{\ln(a)}$, then $\log_a(n) = c\log_b(n)$, which proves that $\log_a(n)$ and $\log_b(n)$ are multiples of each other. Thus, $\log_a(n)$ is $O(\log_b n)$

   Note: Because the base of the logarithm is irrelevant in the context of big-O notation, we can always use just one base.

$$\therefore \log_a(n) \text{ is } O(\text{lg}n).$$

   For any positive $a \neq 1$, where $\lg(n)$ is $\log_2(n)$

- **Big-Ω**. The function $f(n)$ is $\Omega(g(n))$ iff $\exists\ c, N \in \mathbb{R}^+ \ \mid\ f(n) \geqslant cg(n)\ \forall\ n \geqslant N$.

In other words, $cg(n)$ is a lower bound on the size of $f(n)$, or, in the long run, f grows at least at the rate of g

There is an interconnection between these two notations expressed by the equivalence

$$f(n) \text{ is } \Omega(g(n)) \text{ iff } g(n) is O(f(n)).$$

There are an infinite number of possible lower bounds for the function f; that is, there is an infinite set of gs such that $f(n)$ is $\Omega(g(n))$ as well as an unbounded number of possible upper bounds of f. This may be somewhat disquieting, so we restrict our attention to the smallest upper bounds and the largest lower bounds. Note that there is a common ground for big-O and Ω notations indicated by the equalities in the definitions of these notations: Big-O is defined in terms of "$\leqslant$" and Ω in terms of "$\geqslant$"; "=" is included in both inequalities. This suggests a way of restricting the sets of possible lower and upper bounds.

- **Big-Θ**: $f(n)$ is $\Theta(g(n))$ iff $\exists\ c_1, c_2, N \in \mathbb{R}^+ \ \mid\ c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)\ \forall\ n \geqslant N$

  We see that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

  When applying any of these notations, do not forget that they are approximations that hide some detail that in many cases may be considered important.

- **Double O notation**: f is $OO(g(n))$ if it is $O(g(n))$ and the constant c is too large to have practical significance. Thus, $10^8 n$ is $OO(n)$. However, the definition of "too large" depends on the particular application.

- **Using asymptotic complexity to estimate time**: If an algorithm is $O(n^2)$, the time to process n elements is proportional to n^2.

  Let $T(n)$ represent the time, so $T(n) = k \cdot n^2$ where k is a constant.

  To find the time for 1 million elements ($n = 10^6$):

  $$T(10^6) = k \cdot (10^6)^2 = k \cdot 10^{12}$$

  For example, if processing 1000 elements takes 1 second, then:

  $$T(1000) = k \cdot 1000^2 = k \cdot 10^6 \implies k = \frac{1}{10^6}$$

  Now, for $n = 10^6$:

  $$T(10^6) = \frac{1}{10^6} \cdot (10^6)^2 = 10^6 \text{ seconds} = 1,000,000 \text{ seconds} \approx 11.57 \text{ days}.$$

- **Finding asymptotic complexites**: Asymptotic bounds are used to estimate the efficiency of algorithms by assessing the amount of time and memory needed to accomplish the task for which the algorithms were designed. This section illustrates how this complexity can be determined. In most cases, we are interested in time complexity, which usually measures the number of assignments and comparisons performed during the execution of a program. For now let's focus on assignments

  Consider a simple loop to calculate the sum of numbers in an array

```
for (i = sum = 0; i < n; i++)
    sum += a[i];
```

First, two variables are initialized, then the for loop iterates n times, and during each iteration, it executes two assignments, one of which updates sum and the other of which updates i. Thus, there are $2 + 2n$ assignments for the complete run of this for loop; its asymptotic complexity is $O(n)$.

Complexity usually grows if nested loops are used, as in the following code, which outputs the sums of all the subarrays that begin with position 0:

```
for (i = 0; i < n; i++) {
    for (j = 1, sum = a[0]; j <= i; j++)
        sum += a[j];
    cout<<"sum for subarray 0 through "<< i <<" is "<<sum<<endl;
}
```

Before the loops start, i is initialized. The outer loop is performed n times, executing in each iteration an inner `for` loop, print statement, and assignment statements for i, j, and `sum`. The inner loop is executed i times for each $i \in \{1, \ldots, n-1\}$ with two assignments in each iteration: one for `sum` and one for j. Therefore, there are

$$1 + 3n + \sum_{i=1}^{n-1} 2i = 1 + 3n + 2(1 + 2 + \cdots + n - 1) = 1 + 3n + n(n-1)$$

$= O(n) + O(n^2) = O(n^2)$ assignments executed before the program is completed.

- **Amortized complexity**: amortized analysis can be used to find the average complexity of a worst case sequence of operations

Linked lists

2.1 Singly-linked lists

If a node contains a data member that is a pointer to another node, then many nodes can be strung together using only one variable to access the entire sequence of nodes. Such a sequence of nodes is the most frequently used implementation of a linked list, which is a data structure composed of nodes, each node holding some information and a pointer to another node in the list. If a node has a link only to its successor in this sequence, the list is called a singly linked list

Each node resides on the heap

Linked lists can easily grow and shrink in size without reallocating memory or moving elements. Adding or removing nodes (especially at the beginning or middle) is more efficient compared to arrays, as no shifting of elements is required. Memory is allocated as needed, avoiding wasted space typical in arrays with fixed sizes.

However, each node requires extra memory for the pointer to the next node. Accessing elements requires traversal from the head, making lookups slower (O(n)) compared to arrays, which offer O(1) access via indexing. Nodes are scattered in memory, leading to poor cache performance compared to arrays, which have contiguous memory locations.

2.1.1 Structure of the node

The node structure is typically implemented in the following way

```
struct node {
    node* next = nullptr;
    T data = 0;

    node() = default;
    node(data) : data(data) {}
    node(next, data) : next(next), data(data) {}
}
```

A node includes two data members: info and next. The info member is used to store information, and this member is important to the user. The next member is used to link nodes to form a linked list. It is an auxiliary data member used to maintain the linked list. It is indispensable for implementation of the linked list, but less important (if at all) from the user's perspective. Note that node is defined in terms of itself because one data member, next, is a pointer to a node of the same type that is just being defined. Objects that include such a data member are called self-referential objects.

2.1.2 The list class/struct

We also implement the list structure as a class or struct.

```
class single_list {
    node* head = nullptr;
public:
    ...
};
```

2.1.3 Interface of a singly linked list stack

The interface typically includes the following operations:

1. **Insert:** Add a node at the beginning, end, or a specific position in the list.
2. **Delete:** Remove a node from the beginning, end, or a specific position.
3. **Search:** Find a node with a given value.
4. **Traverse:** Iterate through the list to access or print each node's data.
5. **IsEmpty:** Check if the list is empty.
6. **Size:** Return the number of nodes in the list. The first node is called the head, and the last node points to nullptr (indicating the end of the list).

2.1.4 Traversing

Traversing a list is simple.

```
node* curr = head;

while (curr) {
    curr = curr->next;
    ...
}
```

2.1.5 Printing

Now that we can traverse, we can print each node

```
node* curr = head;
while (curr) {
    cout << curr->data;
    curr=curr->next;
}
```

2.1.6 Printing in reverse

Printing in reverse requires creating a stack.

```
if (!head) return; // noop, dont even bother creating a vector.

vector<node*> stack;
node* curr = head;

while (curr) {
    stack.push_back(curr);
    curr=curr->next;
}

for (int i=(int)stack.size()-1; i>=0; --i) {
    cout << stack[i]->data << " ";
}
cout << endl;
```

2.1.7 Getting the length

While we traverse, just increment a counter.

```
size_t len() {
    size_t len = 0;
    for (node* curr = head; curr; curr=curr->next, ++len);
    return len;
}
```

2.1.8 Clearing

```
void clear() {
    node* curr=head, *prev=nullptr;

    while (curr) {
        prev=curr;
        curr=curr->next;
        delete prev;
    }
    head = nullptr;
}
```

2.1.9 Reversing

Reversing is pretty straight forward

```
void reverse() {
    node* prev=nullptr, *curr=head, *next=nullptr;

    while(curr) {
        next=curr->next;
        curr->next = prev;
        prev = curr;
        curr=next;
    }

    head = prev;
}
```

In each iteration, next temporarily holds the next node so you don't lose track of it when reversing the link.

The curr->next pointer is set to prev, effectively reversing the link.

Prev is then updated to curr, and curr is updated to next to continue the process.

2.1.10 Pushing

```cpp
void push(int element) {
    if (!head) {
        head = new node(element);
        return;
    }

    node* curr = head;
    while (curr->next) {
        curr=curr->next;
    }
    curr->next = new node(element);
}
```

2.1.11 Inserting

```cpp
void insert(int pos, int element) {
    if (!head || pos == 0) {
        node* new_node = new node(element);
        new_node->next = head;
        head = new_node;
        return;
    }
    node* curr = head;

    int count=0;
    while (count != pos-1 && curr->next) {
        curr=curr->next;
        ++count;
    }
    node* new_node = new node(element);

    new_node->next = curr->next;
    curr->next = new_node;
}
```

1. **Check if the list is empty or inserting at the head (position 0):**
    - If head is nullptr (meaning the list is empty) or pos == 0 (you want to insert at the beginning), a new node is created with the given element.
    - The new node's next pointer is set to the current head (which could be nullptr if the list is empty), and then head is updated to point to this new node.
    - This handles the case where the new node becomes the first node in the list.
2. **Traverse to the correct position:**
    - If you are inserting somewhere other than the head, the function uses a loop to find the node just before the desired position (pos - 1).
    - It starts at the head and moves along the list until it reaches the node right before where the new node will be inserted.
3. **Insert the new node:**
    - Once the loop finds the right place (curr points to the node before the insertion position), a new node is created.
    - The new node's next pointer is set to curr->next (the node currently in the target position).
    - Then, curr->next is updated to point to the new node, effectively inserting the new node into the list.

2.1.12 Popping

```cpp
void pop() {
    if (!head) return;
    if (!head->next) {
        delete head;
        head=nullptr;
        return;
    }

    node* prev=nullptr, *curr = head;
    while (curr->next) {
        prev=curr;
        curr=curr->next;
    }
    delete curr;
    prev->next=nullptr;
}
```

1. **Empty List Check:** If the list is empty (head == nullptr), it does nothing.

2. **Single Node Case:** If the list has only one node, it deletes the head and sets head to nullptr.

3. **Multiple Nodes:** It traverses to the last node using two pointers (prev and curr), deletes the last node (curr), and sets the second-to-last node's next pointer (prev->next) to nullptr to mark the new end of the list.

2.1.13 Erasing

```cpp
void erase(int element) {
    if (!head) return;

    while (head->data == element) {
        if (head->next && head->data == element) {
            node* tmp = head;
            head = head->next;
            delete tmp;
        }
    }

    node* prev=nullptr, *curr=head;

    while (curr) {
        if (curr->data == element) {
            node* tmp = curr;
            prev->next = curr->next;
            curr=curr->next;
            delete tmp;
        } else {
            prev=curr;
            curr=curr->next;
        }
    }
}
```

This erase function removes all nodes with a specific value (element) from the list:

- **Empty List Check:** If the list is empty (head == nullptr), it returns immediately.
- **Head Node Deletion:** If the head contains the target value, it deletes the head and updates it to the next node. We keep doing this until the head node no longer contains the data we want to remove
- **Traverse and Delete:** It iterates through the list, and for each node with the target value, it removes the node by adjusting the next pointer of the previous node and deleting the current node.

2.1.14 Searching

```cpp
node* search(int element) {
    node* curr = head;
    while (curr) {
        if (curr->data == element) {
            return curr;
        }
    }
    return nullptr;
}
```

Recursion

3.1 Recursion vs iteration

In theory, any problem that can be solved recursively can be solved iteratively. This also means that any problem that can be solved iteratively can also be solved recursively.

The question is, for any problem that can be solved, which method can be used such that the problem is easier to solve.

3.2 Elementary recursion

A recursive definition consists of two parts. In the first part, called the anchor or the ground case, the basic elements that are the building blocks of all other elements of the set are listed. In the second part, rules are given that allow for the construction of new objects out of basic elements or objects that have already been constructed. These rules are applied again and again to generate new objects. For example, to construct the set of natural numbers, one basic element, 0, is singled out, and the operation of incrementing by 1 is given as:

1. $0 \in \mathbb{N}$
2. If $n \in \mathbb{N},\ then(n + 1) \in \mathbb{N}$
3. There are no other objects in the set $\mathbb{N}$

It is more convenient to use the following definition, which encompasses the whole range of Arabic numeric heritage:

1. $0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \in \mathbb{N}$
2. If $n \in \mathbb{N}$, then $n0, n1, n2, n3, n4, n5, n6, n7, n8, n9 \in \mathbb{N}$
3. These are the only natural numbers

Recursive definitions serve two purposes: generating new elements, as already indicated, and testing whether an element belongs to a set. In the case of testing, the problem is solved by reducing it to a simpler problem, and if the simpler problem is still too complex it is reduced to an even simpler problem, and so on, until it is reduced to a problem indicated in the anchor

3.3 Base cases

In recursion, a base case is a condition that stops further recursive calls and provides a direct answer without further recursion

If there were no base case, there would be nothing to stop the recursion. Thus, it would go on until the program crashes. For this reason, all recursive functions must have at least one base case.

If a base case in a recursive function returns a value, then every recursive call leading up to that base case should also return a value. This is necessary to ensure that the result of the recursion is propagated back up the call stack.

In a recursive function, the base case stops the recursion, and if the base case returns something (e.g., a node pointer, integer, etc.), the recursive calls that occur before reaching the base case need to return that result so it can propagate back to the original caller.

3.3.1 Factorials

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n \neq 0 \end{cases}.$$

```cpp
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n-1);

    // Expands to
    // n * n-1 * n-2 * ... * 1
}
```

3.3.2 Powers

Consider the recursive definition for a power of x

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{if } n > 0 \end{cases}.$$

```cpp
constexpr int power(int x, int n) {
    if (n == 0) return 1;
    return x * power(x,n-1);
}
```

The function power() can be implemented differently, without using any recursion, as in the following loop:

```cpp
int power2(int x, int n) {
    int res = 1;

    for (res = x; n > 1; --n) {
        res*=x;
    }
    return res;
}
```

Do we gain anything by using recursion instead of a loop? The recursive version seems to be more intuitive because it is similar to the original definition of the power function. The definition is simply expressed in C++ without losing the original structure of the definition. The recursive version increases program readability, improves self-documentation, and simplifies coding. In our example, the code of the nonrecursive version is not substantially larger than in the recursive version, but for most recursive implementations, the code is shorter than it is in the nonrecursive implementations

3.4 Tail recursion

Tail recursion is a type of recursion where the recursive call is the last thing the function does before returning a result. This means there are no more computations or operations to perform after the recursive call.

Because of this, tail recursion can be optimized by some compilers or interpreters to avoid adding new frames to the call stack, making it more memory-efficient than regular recursion.

In simple terms, if a recursive function calls itself, and after that call there's nothing left to do, it's tail recursion. This allows the function to reuse the same memory space, preventing stack overflow in cases with deep recursion.

the recursive call is not only the last statement but there are no earlier recursive calls, direct or indirect. For example, the function tail() defined as

```
void tail(int i) {
    if (i > 0) {
        cout << i << '';
        tail(i-1);
    }
}
```

Is an example of a function with tail recursion, whereas the function nonTail() defined as

```
void nonTail(int i) {
    if (i > 0) {
        nonTail(i-1);
        cout << i << '';
        nonTail(i-1);
    }
}
```

Is not. Tail recursion is simply a glorified loop and can be easily replaced by one. In this example, it is replaced by substituting a loop for the if statement and decrementing the variable i in accordance with the level of recursive call. In this way, tail() can be expressed by an iterative function:

```
void iterativeEquivalentOfTail(int i) {
    for ( ; i > 0; i--)
    cout << i << '';
}
```

Is there any advantage in using tail recursion over iteration? For languages such as C++, there may be no compelling advantage, but in a language such as Prolog, which has no explicit loop construct (loops are simulated by recursion), tail recursion acquires a much greater weight. In languages endowed with a loop or its equivalents, such as an if statement combined with a goto statement, tail recursion should not be used.

Another problem that can be implemented in recursion is printing an input line in reverse order. Here is a simple recursive implementation:

```
void reverse() {
    char ch;
    cin.get(ch);
    if (ch != '\n') {
        reverse();
        cout.put(ch);
    }
}
```

Compare the recursive implementation with a nonrecursive version of the same function:

```
void simpleIterativeReverse() {
    char stack[80];
    int top = 0;
    cin.getline(stack,80);
    for (top = strlen(stack) - 1; top >= 0; cout.put(stack[top--]));
}
```

functions like strlen() and getline() from the standard C++ library can be used. If we are not supplied with such functions, then our iterative function has to be implemented differently:

```
void iterativeReverse() {
    char stack[80];

    register int top = 0;
    cin.get(stack[top]);

    while(stack[top]!='\n') {
        cin.get(stack[++top]);
    }
    for (top -= 2; top >= 0; cout.put(stack[top--]));
}
```

3.5 Indirect Recursion

The preceding sections discussed only direct recursion, where a function $f()$ called itself. However, $f()$ can call itself indirectly via a chain of other calls. For example, $f()$ can call $g()$, and $g()$ can call $f()$. This is the simplest case of indirect recursion. The chain of intermediate calls can be of an arbitrary length, as in:

$$f() \rightarrow f_1() \rightarrow f_2() \rightarrow ... \rightarrow f_n() \rightarrow f().$$

There is also the situation when $f()$ can call itself indirectly through different chains. Thus, in addition to the chain just given, another chain might also be possible. For instance

$$f() \rightarrow g_1() \rightarrow g_2() \rightarrow ... \rightarrow g_m() \rightarrow f().$$

This situation can be exemplified by three functions used for decoding information. receive() stores the incoming information in a buffer, decode() converts it into legible form, and store() stores it in a file. receive() fills the buffer and calls decode(), which in turn, after finishing its job, submits the buffer with decoded information to store(). After store() accomplishes its tasks, it calls receive() to intercept more encoded information using the same buffer. Therefore, we have the chain of calls

$$\text{recieve()} \rightarrow \text{decode()} \rightarrow \text{store()} \rightarrow \text{recieve()} \rightarrow \text{decode()} \rightarrow ....$$

3.6 Nested Recursion

A more complicated case of recursion is found in definitions in which a function is not only defined in terms of itself, but also is used as one of the parameters. The following definition is an example of such a nesting

$$h(n) = \begin{cases} 0 & \text{if } n = 0 \\ n & \text{if } n > 4 \\ h(2 + h(n)) & \text{if } n \leqslant 4 \end{cases} .$$

3.7 Excessive Recursion

Logical simplicity and readability are used as an argument supporting the use of recursion. The price for using recursion is slowing down execution time and storing on the run-time stack more things than required in a nonrecursive approach. If recursion is too deep (for example, computing $5.6^{100{,}000}$), then we can run out of space on the stack and our program crashes. But usually, the number of recursive calls is much smaller than 100,000, so the danger of overflowing the stack may not be imminent

However, if some recursive function repeats the computations for some parameters, the run time can be prohibitively long even for very simple cases

Consider Fibonacci numbers. A sequence of Fibonacci numbers is defined as follows:

$$\mathrm{Fib}(n) = \begin{cases} n & \text{if } n < 2 \\ \mathrm{Fib}(n-2) + \mathrm{Fib}(n-1) & \text{otherwise} \end{cases} .$$

The definition states that if the first two numbers are 0 and 1, then any number in the sequence is the sum of its two predecessors. But these predecessors are in turn sums of their predecessors, and so on, to the beginning of the sequence.

How can this definition be implemented in C++? It takes almost term-by-term translation to have a recursive version, which is

```
constexpr unsigned long fib(int n) {
    if (n < 2) return n;
    return fib(n-2) + fib(n-1);
}
```

The function is simple and easy to understand but extremely inefficient. To see it, compute Fib(6), the seventh number of the sequence, which is 8. Based on the definition, the computation runs as follows:

$$\begin{aligned} Fib(6) &= Fib(4) + Fib(5) \\ &= Fib(2) + Fib(3) + Fib(5) \\ &= Fib(0) + Fib(1) + Fib(3) + Fib(5) \\ &= 0 + 1 + Fib(3) + Fib(5) \\ &= 1 + Fib(1) + Fib(2) + Fib(5) \\ &= 1 + Fib(1) + Fib(0) + Fib(1) + Fib(5). \end{aligned}$$

Etc... The source of this inefficiency is the repetition of the same calculations because the system forgets what has already been calculated. For example, Fib() is called eight times with parameter n = 1 to decide that 1 can be returned. For each number of the sequence, the function computes all its predecessors without taking into account that it suffices to do this only once.

It takes almost a quarter of a million calls to find the twenty-sixth Fibonacci number, and nearly 3 million calls to determine the thirty-first! This is too heavy a price for the simplicity of the recursive algorithm. As the number of calls and the run time grow exponentially with n, the algorithm has to be abandoned except for very small numbers

An iterative algorithm may be produced rather easily as follows:

```
unsigned long iterativeFib(unsigned long n) {
    if (n < 2)
    return n;
    else {
        register long i = 2, tmp, current = 1, last = 0;
        for ( ; i <= n; ++i) {
            tmp = current;
            current += last;
            last = tmp;
        }
        return current;
    }
}
```

However, there is another, numerical method for computing Fib(n), using a formula discovered by Abraham de Moivre:

$$\mathrm{Fib}(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}.$$

Where $\phi = \frac{1}{2}(1 + \sqrt{5})$, and $\hat{\phi} = 1 - \phi = \frac{1}{2}(1 - \sqrt{5})$. $\hat{\phi}$ becomes very small when n grows, thus it can be omitted.

$$\mathrm{Fib}(n) = \frac{\phi^n}{\sqrt{5}}.$$

Approximated to the nearest integer

```
unsigned long deMoivreFib(unsigned long n) {
    return ceil(exp(n*log(1.6180339897) - log(2.2360679775)) -
↪   .5);
}
```

3.8 Backtracking

In solving some problems, a situation arises where there are different ways leading from a given position, none of them known to lead to a solution. After trying one path unsuccessfully, we return to this crossroads and try to find a solution using another path. However, we must ensure that such a return is possible and that all paths can be tried. This technique is called backtracking, and it allows us to systematically try all available avenues from a certain point after some of them lead to nowhere. Using backtracking, we can always return to a position that offers other possibilities for successfully solving the problem. This technique is used in artificial intelligence, and one of the problems in which backtracking is very useful is the eight queens problem.

The eight queens problem attempts to place eight queens on a chessboard in such a way that no queen is attacking any other To solve this problem, we try to put the first queen on the board, then the second so that it cannot take the first, then the third so that it is not in conflict with the two already placed, and so on, until all of the queens are placed. What happens if, for instance, the sixth queen cannot be placed in a nonconflicting position? We choose another position for the fifth queen and try again with the sixth. If this does not work, the fifth queen is moved again. If all the possible positions for the fifth queen have been tried, the fourth queen is moved and then the process restarts. This process requires a great deal of effort, most of which is spent backtracking to the first crossroads offering some untried avenues. In terms of code, however, the process is rather simple due to the power of recursion, which is a natural implementation of backtracking

```
putQueen(row)
    for every position col on the same row
        if position col is available
            place the next queen in position col;
            if (row < 8)
                putQueen(row+1);
            else success;
            remove the queen from position col;
```

This algorithm finds all possible solutions without regard to the fact that some of them are symmetrical.

3.9 Recursion in singly linked lists

3.9.1 Traversing

To traverse a linked list using recursion, you need to define a recursive function that processes the current node and then calls itself with the next node until the list is fully traversed (i.e., until the current node is nullptr).

```
void TraverseList(node* head) {
    if (!head) {
        return;
    }
    TraverseList(head->next);

    // ...
}
```

3.9.2 Printing

We can use this, for example, to print each nodes data member

```
void PrintList(node* head) {
    if (!head) return;

    cout << head->data << " ";
    PrintList(head->next);
}
```

3.9.3 Printing in reverse

We a slight alter in the print example, we can reverse print the list.

```
void PrintListReverse(node* head) {
    if (!head) return;

    PrintListReverse(head->next);
    cout << head->data << " ";
}
```

3.9.4 Getting the length

3.9.5 Clearing

We can also use this to clear the list

```
void clear() {
    std::function<void(node*)> r_clear = [&] (node* p) = {
        if (!head) return;

        r_clear(head->next);
        delete head;
    }
    r_clear(head);
    head=nullptr;
    size=0;
}
```

3.9.6 Reversing

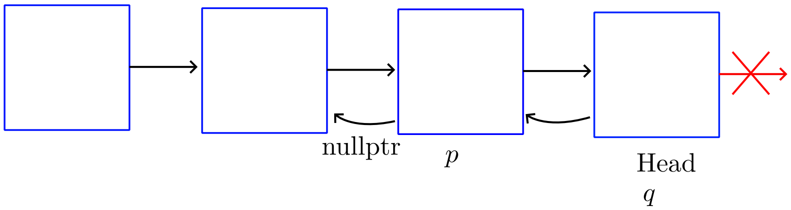
Let's first take a look at the reverse code

```
void reverse() {
    std::function<void(node*)> r_reverse = [&] (node* p) -> void {
        if (!p->next) {
            head = p;
            return;
        }

        r_reverse(p->next);
        node* q = p->next;
        q->next = p;
        p->next = nullptr;

    };
    r_reverse(head);
}
```

The base case is that we are at the end, in this case we set head to this position. Head is now at the end of the list.

Once the base case is triggered and the head is set to the last node in the list, we will be sent back to the n-1 node call.

To get the intuition for linked list logic, we must examine a diagram of the list.



This figure shows the three operations done after each recursive call. In the figure above, we are at the node after the call that set the end to head. We

1. Get a pointer to the node ahead of the current (q).
2. This allows us to severe its old next pointer and reverse its direction.
3. Then, set p next to nullptr (set up for next return).

When the callstack returns to the first call, and does its operations, the list will be reversed

It is also a good idea to examine the iterative method.

```
void Itreverse() {
    node* prev=nullptr, *curr=head, *next=nullptr;

    while (curr) {
        next=curr->next; // Move next to the next node
        curr->next=prev; // Change the direction of current nodes next pointer

        prev=curr; // Advance prev
        curr=next; // Advance curr
    }
    head=prev; // Prev is last node, set head to end
}
```

3.9.7 Pushing

```cpp
void push(int data) {
    if (!head) {
        head = new node(nullptr, data);
        return;
    }
    std::function<void(node*, int)> r_push = [&] (node* curr, int data) -> void {

        if (!curr->next) {
            curr->next = new node(nullptr, data);
            ++size;
            return;
        }
        r_push(curr->next, data);
    };
    r_push(head, data);
}
```

Base case:

1. **Empty list**: No recursion, make head the new node

Otherwise, recurse from the head until we get to the last node, simply set last nodes next pointer to new node and return

3.9.8 Inserting

```
void insert(unsigned pos, int element) {
    std::function<void(node*&, unsigned)> r_insert = [&] (node*& p, unsigned curr_pos) {
        if (curr_pos == 0) {
            node* new_node = new node(nullptr, element);
            new_node->next = p;
            p = new_node;
            return;
        }
        r_insert(p->next,  curr_pos-1);
    };
    r_insert(head, pos);
}
```

Base case

1. **Recursed the same number of times as the `pos` arg**: In this case, make a new node, set next to current node in the recursive traversal, set current node to new node.

Otherwise, keep recursing, subtracting one from the curr_pos.

3.9.9 Popping

```
void pop() {
    if (!head) return;

    if (!head->next) {
        delete head;
        head=nullptr;
        return;
    }

    std::function<void(node*)> r_pop = [&] (node* p) -> void {
        if (!p->next->next) {
            delete p->next;
            p->next = nullptr;
            --size;
            return;
        }
        r_pop(p->next);
    };
    r_pop(head);
}
```

Base cases:

1. **Empty list**: Noop
2. **One node (head)**: Delete then reset head

Otherwise, recurse until we are at the second to last node. Then, delete the second to last nodes next node, which is the last node. Set second to last nodes next pointer to nullptr.

3.9.10 Erasing

```
void erase(int element) {
    std::function<void(node*&)> r_erase = [&] (node*& p) -> void {
        if (p == nullptr) {
            return;
        }

        r_erase(p->next);

        if (p->data == element) {
            node* tmp = p;
            p = p->next;
            delete tmp;
        }
    };
    r_erase(head);
}
```

Base case:

1. **Reached the end**: Return, start unwinding

We traverse to the end of the list recursively, once we reach the end the recursion stops and we start unwinding the call stack, going backwards in the list.

For each node, we check if its data is equal to the element, if it is we set this node equal to its next node, then delete.

3.9.11 Searching

```
node* search(int element) {
    std::function<node*(node*)> r_search = [&] (node* p) -> node* {
        if (p == nullptr)  {
            return nullptr;
        }
        if (p->data == element) {
            return p;
        }
        return r_search(p->next);
    };
    return r_search(head);
}
```

Base cases:

1. **Reached the end of the list**: Element is not in list, return nullptr
2. **Found the first node with the element**: Return the node

Otherwise, recurse through the nodes until we hit one of the base cases.

Binary trees

4.1 Terminology

- **Node:** The basic unit of a binary tree, containing data and references to left and right children.
- **Root:** The topmost node in a tree.
- **Child:** A node directly connected to another node when moving away from the root.
- **Descendants**: The descendants of a node are all nodes that come after a given node.
- **Parent:** The node directly above a child node.
- **Grandparents**: The grandparents of a node is all nodes above the parent up to the root.
- **Ancestors**: The ancestors of a node are all the nodes above a node up to the root
- **Leaf:** A node with no children.
- **Branch node**: A non-leaf node is called a branch node
- **Internal Node:** A branch node, a node with at least one child.
- **Subtree:** A tree consisting of a node and its descendants.
- **Height of a node:** The number of edges on the longest path from a node to a leaf.
- **Height of a tree:** The height of the tree is the height of the root
- **Depth:** The number of edges from the root to a node.
- **Depth of a tree**: The depth of a tree is the depth of the deepest node
- **Degree of a node**: The number of subtrees of a node is called the degree of the node. In a binary tree, all nodes have degree 0, 1, or 2.
- **Degree of a binary tree**: The degree of a tree is the maximum degree of a node in the tree. A binary tree is degree 2.

4.2 Type of binary trees

- **Full Binary Tree:** Every internal node has two children, all leaf nodes have zero children. Thus, all nodes are either zero or two, never one.
- **Complete Binary Tree:** All levels, except possibly the last, are fully filled, and all nodes are as far left as possible.
- **Perfect Binary Tree:** A binary tree where all internal nodes have exactly 2 children, and all leaf nodes are at the same level.
- **Balanced Binary Tree:** A binary tree where the height of the left and right subtrees of every node differs by at most one.
- **Degenerate (or pathological) Tree:** A tree where each parent node has only one child, essentially forming a linked list.
- **Skewed Tree:** A special case of a degenerate tree, where all nodes are skewed to the left or right, forming a linear structure.

4.3 Maximum height of a binary tree

The maximum height of a binary tree with n nodes can be as large as $n1$ (in the case of a degenerate or skewed tree where each node has only one child). This is true for any binary tree:

$$h_{\max} = n - 1.$$

Which occurs for degenerate trees.

4.3.1 Minimum height of a binary tree

The minimum height (best case) for a binary tree with n nodes is achieved when the tree is perfectly balanced:

$$h_{\min} = \lfloor \log_2(n) \rfloor.$$

This is because the tree would need to spread nodes evenly across levels

4.3.2 Number of Leaves in a Binary Tree

For any binary tree with n nodes, the number of leaves l satisfies the following relationship:

$$l \leqslant \frac{n+1}{2}.$$

This formula gives the maximum number of leaves, assuming that the tree is full (every internal node has 2 children).

4.3.3 Relationship Between Internal Nodes and Leaves:

In any binary tree, the number of internal nodes i (nodes with at least one child) and the number of leaves l are related as follows:

$$i \leqslant l - 1.$$

4.3.4 Maximum Number of Nodes at Height h

The maximum number of nodes possible at a given height h (where the height is counted from the root as level 0) in a binary tree is:

$$\text{Max nodes at height } h = 2^h.$$

4.3.5 Number of Edges in a Binary Tree:

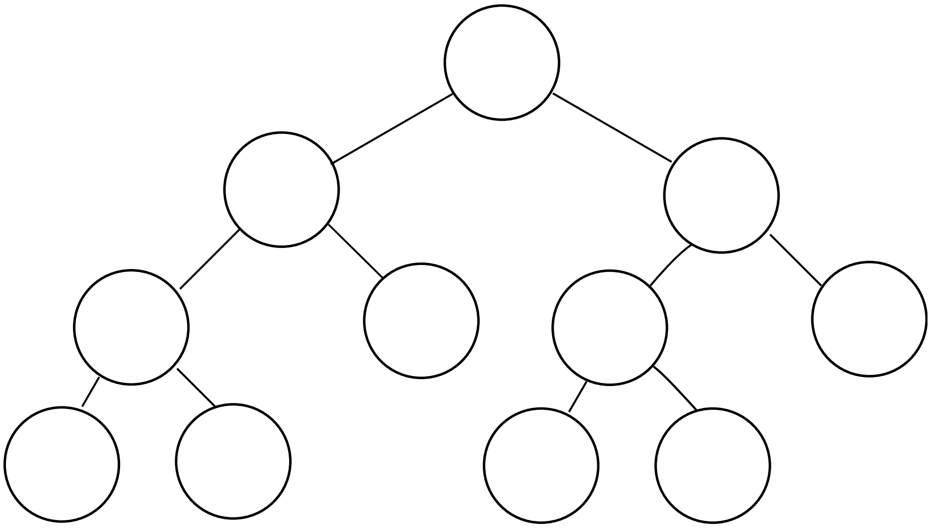
For any binary tree with n nodes, the number of edges e is always

$$e = n - 1.$$

This holds because every node (except the root) is connected to exactly one parent, so there are $n1$ edges in the tree.

4.4 Full trees

A full tree is a tree where all internal nodes are degree two, and all leaf nodes are degree zero. Observe



The next three subsections refer to the *full binary tree theorem*, which states for a nonempty, full tree T

4.4.1 Number of leaves

If T has I internal nodes, the number of leaves is given by

$$L = I + 1.$$

If T has a total of N nodes, the number of leaves is

$$L = \frac{N+1}{2}.$$

4.4.2 Number of nodes

If T has I internal nodes, the total number of nodes is

$$N = 2I + 1.$$

If T has L leaves, the total number of nodes is

$$N = 2L - 1.$$

4.4.3 Number of internal nodes

If T has a total of N nodes, the number of internal nodes is

$$I = \frac{N-1}{2}.$$

If T has L leaves, the number of internal nodes is

$$I = L - 1.$$

4.5 Complete Binary Tree

A complete binary tree has a specific structure defined by how the nodes are filled level by level.

1. **All levels, except possibly the last, are fully filled:**

   In a complete binary tree, every level up to the second-to-last (penultimate) level must be completely filled with nodes. This means that if the tree has height h, levels 0 through $h - 1$ (from the root to the second-to-last level) will have the maximum possible number of nodes for that level.

2. **All nodes are as far left as possible:**

   - On the last level, the nodes don't need to completely fill the level, but the nodes must be positioned as far to the left as possible.
   - For example, if some nodes are missing from the last level, they will always be missing from the right side, not from the left.

Notes: The tree is balanced in terms of node distribution, with all the levels except possibly the last fully filled.

Nodes on the last level are always added from the leftmost position first.

4.5.1 Number of nodes

The height h of a complete binary tree is defined as the number of edges on the longest path from the root to a leaf node.

The total number of nodes in a complete binary tree is given by

$$n = 2^{h+1} - 1.$$

4.5.2 Height

The height h of a complete binary tree with n nodes can be derived as:

$$h = \lfloor \log_2(n) \rfloor.$$

4.5.3 Number of Leaf Nodes (L) in a Complete Binary Tree

The number of leaf nodes in a complete binary tree can be calculated based on the number of internal nodes or the height of the tree

$$L = \lceil \frac{n}{2} \rceil.$$

4.5.4 Number of internal nodes

The number of internal nodes (non-leaf nodes) in a complete binary tree can be calculated as:

$$I = N - L$$

$$I = \lfloor \frac{n}{2} \rfloor.$$

4.5.5 Parent and Child Relationships in a Complete Binary Tree

Parent of node at index i (1-based index):

$$\text{Parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor$$

Left child of node at index i:

$$\text{Left child}(i) = 2i$$

Right child of node at index i:

$$\text{Right child}(i) = 2i + 1$$

These relationships assume a 1-based indexing system for the nodes in the tree (common in heaps or array-based representations).

4.6 Perfect binary tree

4.6.1 Number of Nodes

$$N = 2^{h+1} - 1.$$

4.6.2 Number of Leaf Nodes

$$L = 2^h.$$

4.6.3 Height of the Tree

$$h = \log_2(N + 1) - 1.$$

4.6.4 Number of Internal Nodes

$$I = N - L = 2^h - 1.$$

4.6.5 Depth

$$d = h.$$

Applications of binary trees

5.1 Binary search trees

A binary search tree (BST) is a binary tree in which each node has at most two children and follows these properties:

- **Left Subtree Property:** The value of each node in the left subtree is less than the value of the node itself.
- **Right Subtree Property:** The value of each node in the right subtree is greater than the value of the node itself.
- Both left and right subtrees must also be binary search trees.

5.1.1 Interface

The interface of a Binary Search Tree (BST) typically includes a set of operations for managing and accessing the tree's nodes.

- **Insert(value):** Inserts a new value into the BST while maintaining its properties.
- **Remove(value):** Removes a value from the BST, adjusting the structure to maintain its properties.
- **Predecessor(node)**: Finds the predecessor of a node
- **Succesor(node)**: Finds the successor of a node
- **Find(value):** Searches for a value in the BST and returns the node containing it or null if not found.
- **FindMin():** Returns the node with the smallest value in the BST.
- **FindMax():** Returns the node with the largest value in the BST.
- **IsEmpty():** Checks if the BST is empty.
- **Traverse(order):** Traverses the tree in a specific order (e.g., in-order, pre-order, post-order).
- **Height():** Returns the height of the BST.
- **Clear():** Removes all nodes from the tree, making it empty

5.1.2 Traversals

We can traverse BST's in one of four ways

- Level order
- Preorder
- Inorder
- Postorder

5.1.2.1 Level order

Level-order traversal is a way of visiting all the nodes in a binary tree by levels, from top to bottom. It starts at the root and visits nodes level by level, left to right, for each level.

- Start with the root node (the topmost node).
- Visit all the nodes on the next level (children of the root) from left to right.
- Then, visit all nodes on the level below that (grandchildren of the root) from left to right, and so on.

A queue is often used to implement level-order traversal, as it helps keep track of nodes to visit in the correct order.

```cpp
void levelorderPrint() {
    if (!root) return; // noop for empty tree

    queue<node*> q;
    q.push(root);

    while (!q.empty()) {
        node* curr = q.front();
        q.pop();

        cout << curr->data << endl;
        if (curr->left) {
            q.push(curr->left);
        }
        if (curr->right) {
            q.push(curr->right);
        }
    }
}
```

1. If the list is nonempty, construct a queue and push the root node.
2. While the queue is nonempty, grab the front, process the front, pop the front.
3. Push left and right nodes to queue, if they exist.

5.1.2.2 Preorder

Pre-order traversal is a way of visiting nodes in a binary tree where you:

1. Visit the root node first.
2. Recursively visit the left subtree.
3. Recursively visit the right subtree.

To explain simply:

1. Start with the root node.
2. Go as far left as possible, visiting each node along the way.
3. Once you've reached the end of the left subtree, backtrack and visit the right subtree.

```cpp
void preorderPrint() {
    std::function<void(node*)> r_preorderPrint = [&] (node* p) {
        if (p == nullptr) return;

        cout << p->data << endl;
        r_preorderPrint(p->left);
        r_preorderPrint(p->right);
    };
    r_preorderPrint(root);
}
```

5.1.2.3 Inorder

in-order traversal is a way of visiting nodes in a binary tree where you:

1. Recursively visit the left subtree first.
2. Visit the root node.
3. Recursively visit the right subtree.

To explain simply:

1. Start by going all the way to the left, visiting nodes along the way.
2. Once you reach the leftmost node, visit it, then move up to its parent (the root).
3. After visiting the root, visit the right subtree.

For a BST, printing the tree with an inorder traversal yields a sorted sequence.

```cpp
void inorderPrint() {
    std::function<void(node*)> r_inorderPrint = [&] (node* p) -> void {
        if (!p) return;

        r_inorderPrint(p->left);
        cout << p->data << endl;
        r_inorderPrint(p->right);
    };
    r_inorderPrint(root);
}
```

5.1.2.4 Postorder

Post-order traversal is a way of visiting nodes in a binary tree where you:

1. Recursively visit the left subtree first.
2. Recursively visit the right subtree.
3. Finally, visit the root node.

To explain simply:

1. Start by going to the leftmost node, but don't visit it yet.
2. Then, go to the right subtree and process it.
3. After both subtrees have been visited, visit the root.

```
void postorderPrint() {
    std::function<void(node*)> r_postorderPrint = [&] (node* p) -> void {
        if (!p) return;

        r_postorderPrint(p->left);
        r_postorderPrint(p->right);
        cout << p->data << endl;
    };
    r_postorderPrint(root);
}
```

5.1.3 Successor of a node

The successor of a node is defined mathematically as

$$\mathrm{succ}(X) = \min\{A : \ A > X\}.$$

thus, we find the set of all nodes that have values greater than that of X, then find the minimum in that set.

By properties of binary search trees we find the successor of a node X by

1. **If X has a right child:** The successor is the leftmost node in the right subtree of X (the smallest node in the right subtree).
2. **If X has no right child**:
    - If the node is the left child of its parent, then the parent is its successor.
    - If the node is the right child of its parent, you move upward until you find a node that is the left child of its parent, and that parent is the successor.

5.1.4 Predecessor

The predecessor of a node X is defined as

$$\text{pred}(X) = \max\{A : \ A < X\}.$$

In other words it is the largest node that is less than X. To find the predecessor:

1. **If X has a left child**: The predecssor is the rightmost node in the left subtree
2. **If X has no left child**: The predecessor is the nearest ancestor for which the node is in the right subtree.

5.1.5 The node

The node is similar to a linked list node, but instead of a single next pointer, it has two. A left pointer and a right pointer.

```cpp
struct node{
    node* left = nullptr;
    node* right = nullptr;
    int data = 0;

    node() = default;
    node(int data) : data(data) {}
    node(node* left, node* right, int data) : left(left), right(right), data(data) {}
};
```

5.1.6 The class

For simplicity, we often define the Binary Search Tree (BST) as a class. This allows each instance of the class to hold its own root node, along with other data members such as the size of the tree, that we may need.

If it were not a class, then each function would need to take the root node as an argument and return the (potentially modified) root node to maintain the structure.”

If it were not a class, than each function would have to take as an argument a root node, and return the root node to maintain the structure

```cpp
class BST {
private:
    node* root;
    ...
public:
    ...
};
```

5.1.7 Recursive Insertion

Because of the nature of BSt's, we often use recursion to define the needed operations.

```
void insert(int element)  {
    // If the tree is empty, insert new element as root
    if (!root) {
        root = new node(element);
        return;
    }

    std::function<void(node*)> r_insert = [&](node* p) -> void {

        // If the element is less than current node, and p->left exists, go left
        if (element < p->data && p->left) {
            r_insert(p->left);

            // If the element is greater than current node, and p->right exists, go right
        } else if (element > p->data && p->right) {
            r_insert(p->right);
        }

        // If the element is less than current node, and p->left doesn't exist, insert node as current nodes left child
        if (element < p->data && !p->left) {
            p->left = new node(element);
            return;

            // If the element is greater than current node, and p->right doesn't exist, insert node as current nodes right child
        } else if (element > p->data && !p->right) {
            p->right = new node(element);
            return;
        }
    };
    // Start recursion from the root
    r_insert(root);
}
```

If the tree is empty, it creates a new root node with the given element.

Otherwise, it uses a recursive lambda function (r_insert) to:

- Traverse the tree: going left if the element is smaller, or right if the element is larger.
- Once it finds an appropriate spot (where a left or right child doesn't exist), it inserts the new node as a left or right child accordingly.

The process starts from the root and recursively finds the right place to insert the new element.

5.1.8 Iterative insert

```
void insertB(int element) {
    if (!root) {
        root = new node(element);
        return;
    }

    node* p = root, *trail = nullptr;
    bool left;

    while (p) {
        trail = p;
        if (element < p->data) {
            p=p->left;
            left=true;
        } else if (element > p->data) {
            p=p->right;
            left=false;
        } else {
            return; // noop if already exists
        }
    }
    if (left) {
        trail->left = new node(element);
    } else {
        trail->right = new node(element);
    }
}
```

If the tree is empty, it creates a new root node with the element.

It then iteratively traverses the tree starting from the root:

- Moves left if the element is smaller than the current node's data.
- Moves right if the element is larger.
- If the element already exists, it does nothing and returns.

Once it finds an empty spot (either left or right child is nullptr), it inserts the new node as the left or right child of the parent node (trail), depending on the comparison.

5.1.9 Recursive removing

To remove a node with a given value from a BST, there are three cases

1. Node has no children
2. Node has one child
3. Noe has two children

For case I, we can simply set the nodes parent to nullptr, and then delete the node.

For case II, we must divert the connection from the nodes parent to the nodes child, and then free the node.

Case III is more involved, we first must find the successor of the node. Once we find the successor, we replace the nodes data value with its successor. Then, instead of deleting the node, we delete its successor. Since to be in this case the node must have exactly two children, the successor is found in the simple way.

1. Go right once
2. Go as far left as possible.

Once we have the successor node, it will either have no children, or exactly one child (a right child), if it were to have a left child, it would not be the true successor because we would have not gone as far left as possible.

```cpp
void remove(int element) {
    if (!root) return; // Noop for empty tree

    std::function<void(node*&, node*&)> r_remove = [&] (node*& p, node*& last) -> void {
        if (!p) return; // Not found in tree

        if (element < p->data) {
            r_remove(p->left, p);
        } else if (element > p->data) {
            r_remove(p->right, p);
        } else { // Found
            // Case I: Node has zero children
            if (!p->left && !p->right) {
                node* tmp = p;
                p=nullptr;
                delete tmp;
                // Case II: Node has one child
            } else if (!p->left || !p->right) {
                node* tmp = p;
                p = (p->left ? p->left : p->right);
                delete tmp;
                // Case III: Two children
            } else {
                node* successor = p->right;
                node* successorParent = p;

                // Find the in-order successor
                while (successor->left) {
                    successorParent = successor;
                    successor = successor->left;
                }

                // Replace nodes value with successor value
                p->data = successor->data;

                // Now we need to delete the successor node
                // The successor is a leaf or has a right child
                if (successorParent->left == successor) {
                    successorParent->left = successor->right;
                } else {
                    successorParent->right = successor->right;
                }
                delete successor;
            }
        }
    };
    r_remove(root,root);
}
```

5.1.10 Clearing

```cpp
void clear() {
    if (!root) return;

    std::function<void(node*)> r_clear = [&](node* p) -> void {
        if (!p) return;

        r_clear(p->left);
        r_clear(p->right);

        delete p;
    };
    r_clear(root);
    root = nullptr;
}
```

This function deletes all nodes in a binary search tree. It recursively traverses the tree, deleting each node after its children have been deleted, and finally sets the root to nullptr, effectively clearing the entire tree

5.1.11 Counting the height

```cpp
size_t height() {
    std::function<size_t(node*)> r_height = [&](node* p) -> size_t {
        // Base case height of a nullptr is zero
        if (!p) return 0;
        return 1+std::max(r_height(p->left), r_height(p->right));
    };
    // Height is counting edges, so its number nodes in longest path from root to leaf - 1
    return r_height(root) -1;
}
```

This code defines a height() function that calculates the height of a binary tree by counting the edges. It uses a recursive lambda function r_height to traverse the tree. For each node, it returns 1 + max(left subtree height, right subtree height) to find the longest path from the root to any leaf. Since r_height counts nodes, the function subtracts 1 at the end to convert the node count to edge count, which is the definition of height.

5.1.12 Getting the depth of the node

```
int nodeDepth(node* p) {
    if (p == root) return 1;
    return r_nodeDepth(root, p, 1);
}

int r_nodeDepth(node* curr, node* p, int depth) {
    if (curr == nullptr) {
        return -1;  // Node not found
    }
    if (p == curr) {
        return depth;  // Node found, return the depth
    }

    // Recursively search in the left subtree if p's data is smaller
    if (p->data < curr->data) {
        return r_nodeDepth(curr->left, p, depth + 1);
    }
    // Recursively search in the right subtree if p's data is greater
    if (p->data > curr->data) {
        return r_nodeDepth(curr->right, p, depth + 1);
    }

    return -1;  // This should never be reached if the tree is valid
}
```

This code defines two functions that work together to calculate the depth of a given node p in a binary search tree (BST):

nodeDepth: This function is the entry point to calculate the depth of the node *p*.

It first checks if *p* is the root node. If so, it returns 1 since the root node is considered to have a depth of 1.

If *p* is not the root, it calls the helper function r_nodeDepth to recursively search for the node, starting from the root with an initial depth of 1.

r_nodeDepth: This is a recursive helper function that searches for the node *p* in the tree, while tracking the current depth.

It first checks if curr (the current node in the search) is nullptr, which indicates that the node *p* is not in the tree. In that case, it returns -1.

If curr matches *p*, it returns the current depth.

Otherwise, it recursively searches in the left or right subtree, depending on whether *p*'s data is less than or greater than the current node's data, and increments the depth by 1 at each recursive step.

5.1.13 Degenerate Binary Search trees

A degenerate binary search tree is a tree where each parent node has only one child, causing the tree to resemble a linked list

Consider building a binary search tree, taking values from a sorted array from left to right. Since all subsequent entries will be greater than the previous, the insertions will only go in one direction, right. Thus, the final tree will resemble a linked list and thus we will not be able to use the $\lg(n)$ property of binary trees.

5.1.14 Complexities

Because BST have no guarantee of being well formed, (ie degenerate trees), the complexity of many operations in the worst case is $O(n)$.

Operation	**Best Case**	**Average Case**	**Worst Case**
Insertion	$O(\log n)$	$O(\log n)$	$O(n)$
Search	$O(1)$	$O(\log n)$	$O(n)$
Removal	$O(\log n)$	$O(\log n)$	$O(n)$
Height	$O(\log n)$	—	$O(n)$
Traversal	$O(n)$	$O(n)$	$O(n)$

Note: $\Omega(\lg(n)$ for BST operations like search, insert, and delete (best case).

$\Omega(\lg(n))$ for operations that require visiting all nodes, like traversals.

5.2 Adelson-Velsky and Landis Trees (AVL trees)

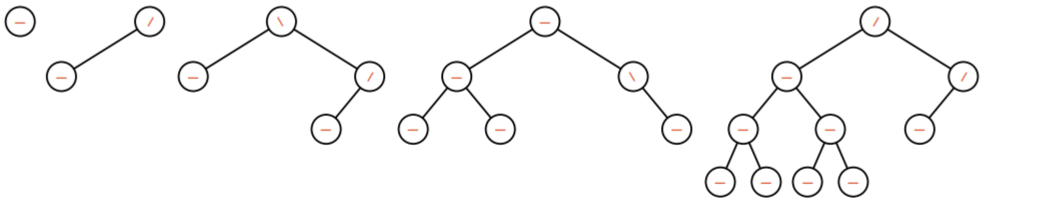
When dealing with binary search trees, insertions and removals occur continually, with no predictable order. In some of these applications, it is important to optimize search times by keeping the tree very nearly balanced at all times. The method in this section for achieving this goal was described in 1962 by two Russian mathematicians, G. M. ADEL’SON-VEL’SKI˘I and E. M. LANDIS, and the resulting binary search trees are called AVL trees in their honor.

AVL trees achieve the goal that searches, insertions, and removals in a tree with n nodes can all be achieved in time that is $O(\log n)$, even in the worst case. The height of an AVL tree with n nodes, as we shall establish, can never exceed lg n, and thus even in the worst case, the behavior of an AVL tree could not be much below that of a random binary search tree. In almost all cases, however, the actual length of a search is very nearly lg n, and thus the behavior of AVL trees closely approximates that of the ideal, completely balanced binary search tree.

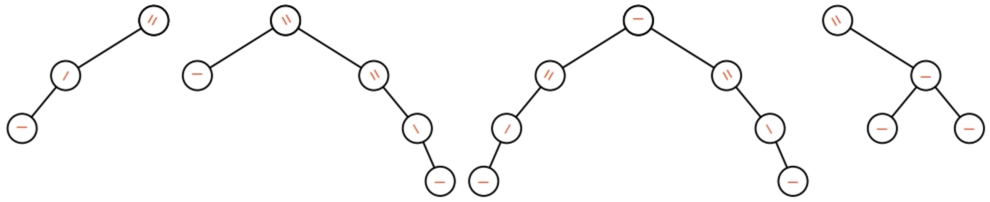
5.2.1 Definition

An AVL tree is a binary search tree in which the heights of the left and right subtrees of the root differ by at most 1 and in which the left and right subtrees are again AVL trees.

With each node of an AVL tree is associated a *balance factor* that is `lefthigher`, `equal-height`, or `right-higher` according, respectively, as the left subtree has height greater than, equal to, or less than that of the right subtree.



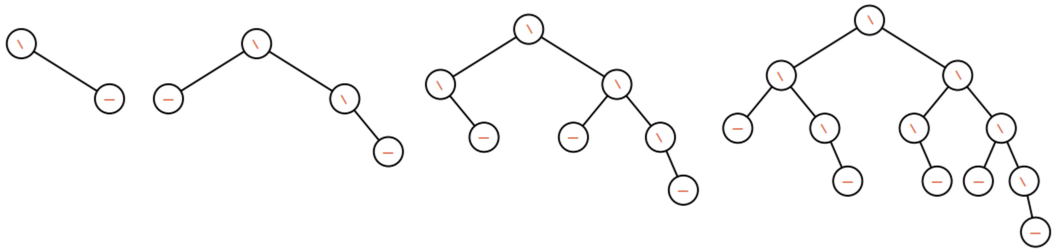
AVL trees



non-AVL trees

In drawing diagrams, we shall show a left-higher node by ‘/,’ a node whose balance 358

factor is equal by ‘–,’ and a right-higher node by ‘’. The figure above shows several small AVL trees, as well as some binary trees that fail to satisfy the definition. Note that the definition does not require that all leaves be on the same or adjacent levels.



The figure above shows several AVL trees that are quite skewed, with right subtrees having greater height than left subtrees.

5.2.2 Defining balance factors in C++

We employ an enumerated data type to record balance factors

```
enum Balance_factor { left_higher, equal_height, right_higher };
```

Balance factors must be included in all the nodes of an AVL tree, and we must adapt our former node specification accordingly.

5.2.3 AVL Nodes

A typical node for an AVL tree is as follows

```
enum balance {
    eh, rh, lh
};

struct node {
    node* left{nullptr}, *right{nullptr};
    int data{0};
    balance b{eh};
    int height{0};

    // Constructors
        ...
};
```

In AVL trees, it is common for the node structure to include a height member. This height field is essential for efficiently maintaining the balance of the tree, as the balance factor of a node (the difference in height between its left and right subtrees) is used to determine whether the tree needs rebalancing after insertions or deletions.

Without the height member, recalculating the height of each node during every operation would require traversing the entire subtree, significantly increasing the time complexity. By storing the height in each node, you can retrieve it in constant time, allowing rotations and rebalancing operations to remain efficient.

5.2.4 Interface

The interface of a Binary Search Tree (BST) and an AVL Tree is generally the same. Both are types of binary trees, and they share similar operations such as:

- **Insert:** Insert a new element into the tree.
- **Remove/Delete:** Remove an element from the tree.
- **Search:** Find whether a particular element exists in the tree.
- **Traversal:** In-order, pre-order, post-order, and level-order traversals.

However, behind the scenes, the AVL tree performs additional work to maintain its balance property, but this doesn't typically change the public interface

5.2.5 Balancing an AVL tree

As we insert or remove nodes from the tree, it may happen that the resulting tree fails to satisfy the conditions imposed by AVL trees. To *rebalance* the tree, we have a set of operations, called rotations. We have

1. **Right Rotation (RR):** Applied when a left subtree is too deep. The subtree is rotated to the right, reducing the height of the left side.
2. **Left Rotation (LL):** Applied when a right subtree is too deep. The subtree is rotated to the left, reducing the height of the right side.
3. **Left-Right Rotation (LR):** Occurs when a left subtree has a deep right subtree. First, a left rotation is performed on the left subtree, followed by a right rotation.
4. **Right-Left Rotation (RL):** Occurs when a right subtree has a deep left subtree. First, a right rotation is performed on the right subtree, followed by a left rotation.

These rotations are applied based on the balance factor, ensuring the tree remains balanced with a height difference of at most 1 between subtrees.

Note: Left-right and Right-left rotations are also called double right and double left respectively.

When writing our rotation algorithms, we only need to define two. A left rotation algorithm and a right rotation algorithm

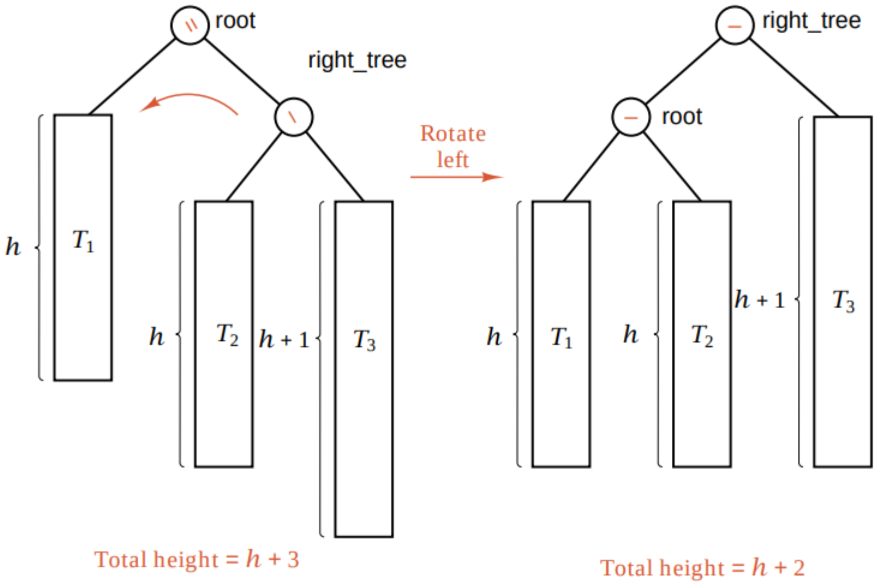
Note: Performing a rotation when the height difference is only 1 would be unnecessary and could actually disrupt the balancing of the tree. The tree is still balanced in this case because a height difference of 1 between subtrees is allowed in AVL trees.

5.2.6 Rotations: Right tree

Let us now consider the case when a new node has been inserted into the taller subtree of a root node and its height has increased, so that now one subtree has height 2 more than the other, and the tree no longer satisfies the AVL requirements. We must now rebuild part of the tree to restore its balance. To be definite, let us assume that we have inserted the new node into the right subtree, its height has increased, and the original tree was right higher. That is, we wish to consider the case covered by the function right_balance. Let root denote the root of the tree and right_tree the root of its right subtree.

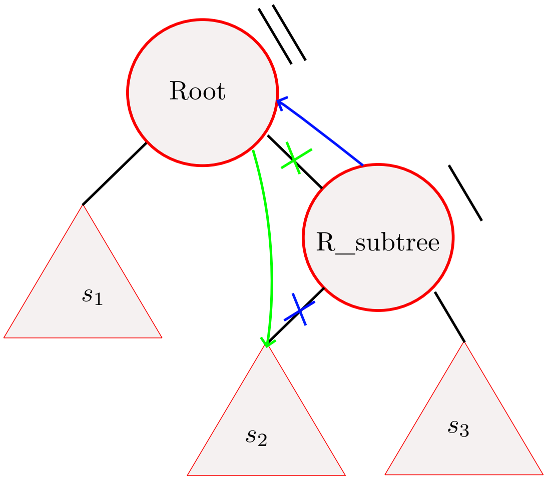
There are three cases to consider, depending on the balance factor of right_tree.

5.2.6.1 Case 1: Right higher The first case, when right_tree is right higher. The action needed in this case is called a left rotation. We have rotated the node right_tree upward to the root, dropping root down into the left subtree of right_tree; the subtree T2 of nodes with keys between those of root and right_tree now becomes the right subtree of root rather than the left subtree of right_tree. A left rotation is succinctly described in the following C++ function. Note especially that, when done in the appropriate order, the steps constitute a rotation of the values in three pointer variables. Note also that, after the rotation, the height of the rotated tree has decreased by 1; it had previously increased because of the insertion; hence the height finishes where it began.



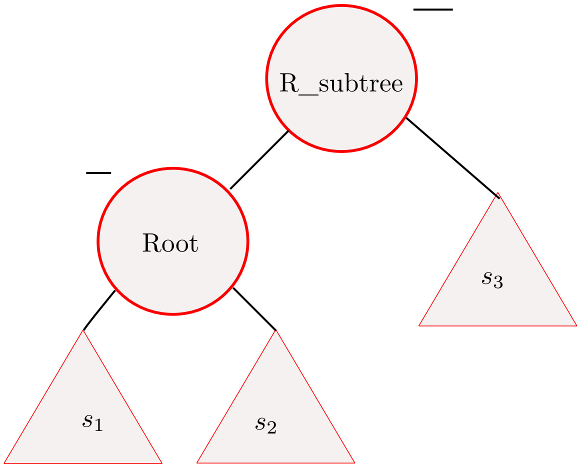
Thus, we come to the following implementation of a left rotation.

```
void left_rotate(node* root) {
    if (!root || !root->right) return;

    right_subtree = root->right;
    root->left = right_subtree->left;
    right_subtree->left = root;
    right_subtree=root;
}
```



1. **Step 1 (green)**: Attach right_subtrees left subtree to the right of root
2. **Step 2 (blue)**: Attach root to the left of right_subtree
3. **Step 3**: Make r_subtree the new root.

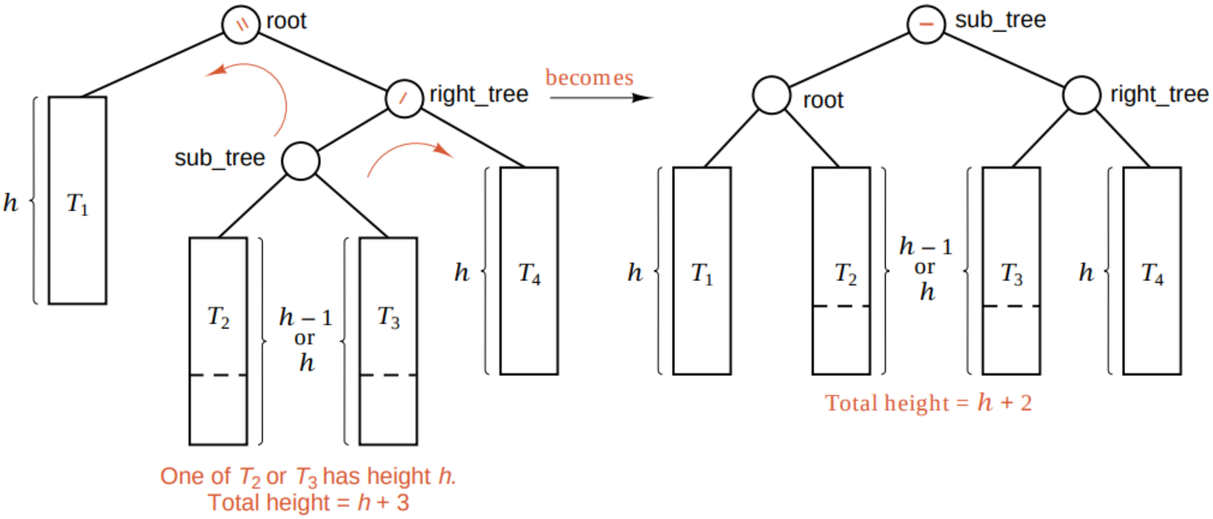
Yields the rotated tree



Note that we perform a left rotation when the roots right subtree has an insertion into its right subtree. Notice that the balance symbols point in the same direction. The only balance factors that change are the balance factors of root and right_subtree, they become even (balanced).

5.2.6.2 Case 2: Left higher

The second case, when the balance factor of right_tree is left higher, is slightly more complicated. It is necessary to move two levels, to the node sub_tree that roots the left subtree of right_tree, to find the new root. This process is shown in Figure 10.20 and is called a double rotation, because the transformation can be obtained in two steps by first rotating the subtree with root right_tree to the right (so that sub_tree becomes its root), and then rotating the tree pointed to by root to the left (moving sub_tree up to become the new root).



We see from the figure that we first perform a right rotation on right_tree, and right_trees left subtree, this shifts the subtree unbalance in right_subtree such that it becomes unbalanced on the right instead of on the left. This enables us to perform a left rotation on root and right_tree. Note that after the right rotation, the balance symbols point in the same direction. A right rotation is of the form

5.2.7 Insertion of a Node

We can insert a new node into an AVL tree by first following the usual binary tree insertion algorithm: comparing the key of the new node with that in the root, and inserting the new node into the left or right subtree as appropriate. It often turns out that the new node can be inserted without changing the height of the subtree, in which case neither the height nor the balance of the root will be changed. Even when the height of a subtree does increase, it may be the shorter subtree that has grown, so only the balance factor of the root will change. The only case that can cause difficulty occurs when the new node is added to a subtree of the root that is strictly taller than the other subtree, and the height is increased. This would cause one subtree to have height 2 more than the other, whereas the AVL condition is that the height difference is never more than 1

The basic structure of our algorithm will thus be the same as the ordinary recursive binary tree insertion algorithm, but with significant additions to accommodate the processing of balance factors and other structure of AVL trees

We must keep track of whether an insertion (after recursion) has increased the tree height or not, so that the balance factors can be changed appropriately. This we do by including an additional calling parameter taller of type bool in the auxiliary recursive function called by the insertion method. The task of restoring balance when required will be done in the subsidiary functions left_balance and right_balance.

Math algorithms

6.0.1 Euclidean GCD Algorithm

The GCD of two integers a and b (with $a \leqslant b$) is the largest integer that divides both a and b. The Euclidean algorithm is based on the principle that

$$\gcd(a, b) = \gcd(b, a\text{mod } b).$$

This means that the GCD of two numbers doesn't change if the larger number is replaced by its remainder when divided by the smaller number. You keep repeating this until the remainder is 0, and the GCD will be the last non-zero remainder

```
int gcd(int a, int b)  {
    if (!b) return a;

    return gcd(b, a%b);
}
```