Algorithm

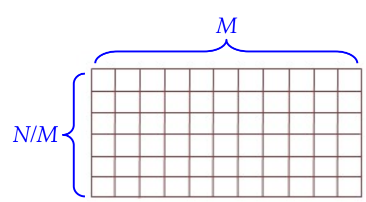
Sample codes in the document haven't been compiled in C++11 IDE

1.	General	6		sorting and searching are related but different things :
2.	Bisection	4	→	sorting is useful for efficient searching, such as binary search
3.	Vector	6	→	sorting is *NOT* used for searching only, for example, we want ordered iterating
4.	Stack and queue	4	→	searching can be made efficient without sorting, for example, hash
5.	Linked list	4		
6.	Binary tree	6		
7.	Binary tree variants	4	→	heap tree / btree / prefix tree / skip list
8.	Graph algorithms	3	→	topological sorting / shortest path / disjoint set (union find algo)
9.	Sorting	8		

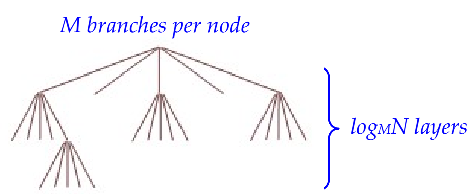
Time complexity

analytic solution	$O(1)$		
bisection	$O(logN)$		
scan once	$O(N)$		
scan once × bisection	$O(NlogN)$	*or*	$O(NlogM)$
scan in 2 dimensions	$O(N^2)$	*or*	$O(NM)$
scan in 2 dimensions × bisection	$O(N^2logN)$	*or*	$O(NMlogK)$
scan in 3 dimensions	$O(N^3)$	*or*	$O(NMK)$

Division finds the number of rows when N items are distributed over M columns.



Logarithm finds the number of layers when N items are distributed over leaves of M-branches tree.



1. General

Question 1 – Hanoi Tower

Lets start with classical Hanoi Tower. Let's call the 3 towers as A, B and C, with A as the source, while B or C can be the destination.

Recursive method

- recursive call to move all discs (except the biggest one) from A to B
- move the biggest disc from A to C
- recursive call to move all discs (except the biggest one) from B to C

Iterative method

- increment position of the smallest disc, from A to B, or from B to C, or from C to A
- ignore the tower on which the smallest disc locates, make one valid move (there is only one possible choice)
- repeat these two steps until the problem is solved

Question 2 – Euclidean algorithm

Greatest common divisor of A and B (suppose $A > B$) equals to the greatest common divisor of (A *mod* B) and B. Now let's prove the Euclidean algorithm. Given that g be the *GCD(A,B)*.

Def	$g = GCD(A,B)$	
$\Rightarrow$	$g \mid A$ and $g \mid B$	*notation for divisible*
$\Rightarrow$	$g \mid nB + r$ and $g \mid B$	*assuming that* $A = nB + r$ *where* $r = A \bmod B$
$\Rightarrow$	$g \mid r$ and $g \mid B$	
$\Rightarrow$	$g \mid GCD(r,B)$	

Conversely, if we assume that g' be *GCB(r,B)*, we have :

Def	$g' = GCD(r,B)$	
$\Rightarrow$	$g' \mid r$ and $g' \mid B$	*notation for divisible*
$\Rightarrow$	$g' \mid nB + r$ and $g' \mid B$	*assuming that* $A = nB + r$ *where* $r = A \bmod B$
$\Rightarrow$	$g' \mid A$ and $g \mid B$	
$\Rightarrow$	$g' \mid GCD(A,B)$	

Hence we have : *GCD(A,B)*|*GCD(r,B)* and *GCD(r,B)*|*GCD(A,B)*, which is possible only if *GCD(A,B) = GCD(r,B)) = GCD(A mod B,B)*.

	basic number theory without proof	
1	$GCD(A,B) = \mathrm{GCD}(A \bmod \mathrm{B}, \mathrm{B})$	*Euclidean algorithm*
2	$GCD(A,B) \times LCM(A,B) = A \times B$	
3	*integer x satisfying* $(x-1)! \bmod x = x-1 \leftrightarrow$ *x is prime number*	*William theorem*
4	*integers x,y, are prime, greatest integer z such that* $z \neq nx + my$ *, then* $z = xy - x - y$	*Chicken McNugget theorem*

Question 3 – How to implement big number?

Represent big number as a string, then implement *(read "Interview - Optiver1.doc")* :

- addition (sum and carry)
- subtraction (differerence and borrow)
- long multiplication and long division

```
// please implement the following
int ctoi(char c) { return c-'0'; }
std::string add(const std::string& x, const std::string& y);
std::string scale(const std::string& x, int scale, int order);
std::string multiply(const std::string& x, const std::string& y);
```

Question 4 – How to reverse an integer?

Lets do it without conversion to string.

```
int reverse(int x)
{
    int y = 0;
    while (x > 0)
    {
        y = y * 10 + x % 10;
        x = x / 10;
    }
    return y;
}
```

```
int reverse_binary(int x)
{
    int y = 0;
    while (x > 0)
    {
        y = y << 1 + x % 2;
        x = x >> 1;
    }
    return y;
}
```

Counting number of bits in binary representation (from Facebook interview).

```
int count_bit(int x)
{

    int y = 0;
    while (x > 0)
    {
        y += x % 2;
        x = x >> 1;
    }
    return y;
}
```

```
int count_bit(int x) // consider more than 2 cases per iteration
{
    LUT int[] = {0,1,1,2};
    int y = 0;
    while (x > 0)
    {
        y += LUT[x % 4];
        x = x >> 2;
    }
    return y;
}
```

Implement (2) integer/integer division (1) integer * interger multiplication with addition / subtraction and bit shift.

```
int multiply(int x, int y)
{



    int z = 0;
    while (y > 0)
    {
        if (y%2 == 1) z += x;
        x = x << 1;
        y = y >> 1;
    }
    return z;
}
```

```
int divide(int x, int y)
{
    int divisor = y;
    while (y <= x) y = y << 1;
    y = y >> 1;

    int z = 0;
    while (y >= divisor)
    {
        if (x >= y) { x = x-y;   z = (z << 1) + 1; }
        else                     z = (z << 1) + 0;
        y = y >> 1;
    }
    return z;
}
```

Question 5 – Next number of digit set (or ***next permutation*** in LeetCode)

Given a set of digits in the form of an integer, for example, 357436521, by rearranging the order of digits, we can get a new integer. Find the minimum integer (after swapping) that is greater than the original one. This is not about dynamic programming. Lets start from basic : swapping of 2 digits yields a greater integer if the more significant digits is smaller than another (and vice versa). In the example, it exhibits an increasing trend (6521) if we start iterating from LSD (least significant digit), hence swapping between any 2 digits in an increasing trend yields a smaller integer. We then move forward to the next more significant digit, i.e. 3, there exists multiple less significant digits which are greater than 3, i.e. 5 or 6, hence a swap between (3,5) or between (3,6) does yield a greater integer, we pick (3,5) because 5 is smaller than 6. Now we have : 357456321, however this is not the minimum solution, we can re-arrange all digits on RHS of 5 in decreasing order (starting from LSD) to yield the next integer, 357451236. Let's do it inplace.

```
bool next_integer(std::string& integer) // Please handle (1) integer.size()==0,1 (2) integer is already max
{
    int edge = -1;

    // step 1 : find the edge (first drop after increasing trend from LSD)
    for(int n=integer.size()-1; n!=0; --n) { if (integer[n-1] < integer[n]) edge = n-1; }
    if (edge < 0) return false; // both case 1,2 exit here

    // step 2 : min that is greater than the edge
    auto min = integer[edge+1];
    auto min_iter = integer.begin()+edge+1;
    for(auto iter = integer.begin()+edge+1; iter!=integer.end(); ++iter)
    {
        if (*iter > integer[edge] && *iter < min) { min = *iter; min_iter = iter; }
    }

    // step 3 : swap the edge with the next greater digit (note = edge is index, iter is iterator)
    std::swap(integer[edge], *min_iter);

    // step 4 : sort RHS of the edge
    std::sort(integer.begin()+edge+1, integer.end());
    return true;
}
```

Question 6 – Translating numbers to strings

Given an integer, translate it to a string, using mapping : 1 to a, 2 to b, …, 10 to j, 11 to k, … and 26 to z. For example, integer 12258 can be translated to "abbeh", "aveh", "abyh", "lbeh" and "lyh", i.e. 5 possible translations. Write a function to count the number of translation, given an integer. Please be reminded that 10 returns 1, 20 returns 1, however 30 returns 0, 100 returns 0, 26 returns 2, 27 returns 1 etc. Lets try a top-down recursive approach.

```
unsigned long count_single_digit(unsigned long x) { if (x > 0   && x < 10) return 1; else return 0; }
unsigned long count_double_digit(unsigned long x) { if (x >= 10 && x <=26) return 1; else return 0; }
unsigned long count_translation(unsigned long x)
{
    if (x < 10)  return count_single_digit(x);
    if (x < 100) return count_double_digit(x) + count_single_digit(x/10) * count_single_digit(x%10);
    return (count_translation(x/10)  * count_single_digit(x%10) +
            count_translation(x/100) * count_double_digit(x%100)); // red indicates recursion
}
```

Question 7 – Count divisibility

Write function `int solution(int A, int B, int K)` which when given three integers *A*, *B* and *K*, returns the number of integers within the range *[A...B]* that are divisible by K, with *O(1)* time and *O(1)* space.

```
int solution(int A, int B, int K)
{
    if (A==0) return B/K+1;
    return B/K-(A-1)/K;
}
```

Question 8 – Cyclic rotation

Rotation is defined as shifting array to RHS by one index, with last element moved to front. Write a function to do K steps rotation.

```
std::vector<int> solution(std::vector<int> &A, int K)
{
    std::vector<int> output;
    output.resize(A.size());
    for(int n=0; n!=A.size(); ++n)
    {
        output[(n+K)%A.size()] = A[n];
    }
    return output;
}
```

Question 9 – Distinct numbers

Count the number of distinct numbers in a vector, vector maximum size is *100K*, while element values lie within *[-1M,+1M]*.

```
int solution(const std::vector<int> &A)
{
    std::unordered_map map;
    for(const auto& x : A)
    {
        if (auto iter = map.find(x); iter == map.end()) map[x] = 1; else ++map[x];
    }

    int count = 0;
    for(const auto& x : map)
    {
        if (x.second == 1) ++count;
    }
    return count;
}
```

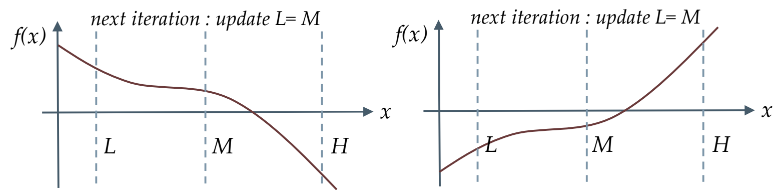
2. Bisection (From continuous function to discrete function)

Bisection for continuous function $f(x) = 0$:

- it works for monotonic function only
- it reduces $O(N)$ linear search to $O(logN)$

Bisection for continuous function is done by maintaining 6 variables $\{L,f(L)\}$ $\{M,f(M)\}$ and $\{H,f(H)\}$:

- initialize L and $y_L = f(L)$
- initialize H and $y_U = f(H)$
- inside while-loop :
- update M and $y_M = f(M)$
- update either $\{L= M, y_L= y_M\}$ or $\{H= M, y_H= y_M\}$



Bisection for discrete function, i.e. a vector of integers, is slightly different from continuous case :

- no need to call `f()` as it can be read directly from input vector
- iteration stop-condition for integers should take care of the fact that : $(n+(n+1)) / 2$ becomes n
- iteration update-equation should be done by bisection, avoid trisection

Question 1

Bisection for continuous function.

```
std::optional<double> bisect(std::function<double(double)> f, double x0, double x1)
{
    double y0 = f(x0);                                  // 1. initial value
    double y1 = f(x1);

    if (y0 * y1 > 0) return std::nullopt;               // 2. check no solution
    if (y0 > 0)      return bisect(f,x1,x0);

    while(x1-x0 > tolerance)                            // 3. check continue
    {
        double xm = (x0+x1)/2;                          // 4. mid point
        double ym = f(xm);
        if (ym < 0) { x0 = xm; y0 = ym; }               // 5. iterative update, with 2 cases
        else        { x1 = xm; y1 = ym; }
    }

    return std::make_optional((x0+x1)/2);               // 6. final output
}
```

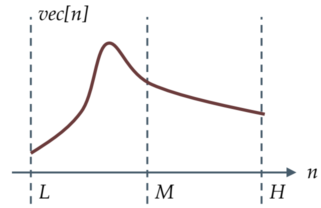
Question 2

Given a sorted (either ascending or descending) signed-integer array, find the element that equals to a target.

```
std::optional<uint32_t> equals_to(const vector<int32_t>& vec, int32_t target)
{
    // 1. check size and init x0 & x1
    if (vec.size()==0)              return std::nullopt;
    if (vec.size()==1)              return (vec[0]==target? std::make_optional(x0) : std::nullopt);
    if (vec.front() < vec.back())   return equals_to_impl(vec, target, 0, vec.size()-1);
    else                            return equals_to_impl(vec, target, vec.size()-1, 0);
}

std::optional<uint32_t> equals_to_impl(const vector<int32_t>& vec, int32_t target, uint32_t x0, uint32_t x1)
{
    if (vec[x0] > target)  return std::nullopt;         // 2. check no solution
    if (vec[x1] < target)  return std::nullopt;

    while(abs(x1-x0)>1)                                     // 3. check continue
    {
        std::uint32_t xm = (x0+x1) >> 1;                // 4. mid point
        if (vec[xm] < target)       x0 = xm;            // 5. iterative update, with 2 cases
        else                        x1 = xm;
    }

    if (vec[x0] == target) return std::make_optional(x0);   // 6. final output
    if (vec[x1] == target) return std::make_optional(x1);
    return std::nullopt;
}
```

Question 3

Given a unimodal integer array, find the turning point. Unimodal array is increasing sequence followed by a decreasing sequence, hence this is a one dimensional maximization problem. Return null for array `[1,2,3,3,3,3,2,1]`.

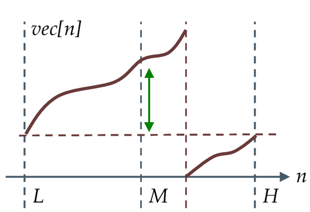


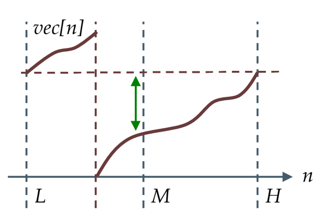

```
std::optional<int32_t> turning_point(const vector<int32_t>& vec)
{
    // 1. check size and init x0 & x1
    if (vec.size() <= 2) return std::nullopt;
    if (vec.size() == 3) return (is_peak(vec,1)? std::make_optional(1) : std::nullopt);
    std::uint32_t x0 = 1;
    std::uint32_t x1 = vec.size()-2;
                                                    // 2. (skip) check no solution
    while(x0!=x1-1)                                 // 3. check continue
    {
        std:uint32_t xm = (x0+x1) >> 1;             // 4. mid point
        if (uptrend(vec, xm))   x0 = xm;            // 5. iterative update, with 2 cases
        else                    x1 = xm;
    }

    if (is_peak(vec,x0))    return std::make_optional(x0);   // 6. final output
    if (is_peak(vec,x1))    return std::make_optional(x1);
    return std::nullopt;
}

bool is_peak(const vector<uint32_t>& vec, uint32_t x) { return vec[n] > vec[n-1] && vec[n] > vec[n+1]; } // x = [1,N-2]
bool uptrend(const vector<uint32_t>& vec, uint32_t x) { return vec[n-1] < vec[n] && vec[n] < vec[n+1]; } // x = [1,N-2]
```

Question 4

Given a rotated integer array, find a specified integer. The rotation of a vector is applying a sequence of "pop-front-and-push-back" operations on a vector. The implementation inside the while-loop is like a two-layer decision tree.

```
std::optional<std::int32_t> search_rotated_vector(const std::vector<std::int32_t>& vec, std::int32_t target)
{
    // 1. check size and init x0 & x1
    if (vec.size()==0) return std::nullopt;
    if (vec.size()==1) return (vec[0]==target? std::make_optional(x0) : std::nullopt);
    std::uint32_t x0 = 0;
    std::uint32_t x1 = vec.size()-1;
                                                    // 2. (skip) check no solution
    while(x0!=x1-1)                                 // 3. check continue
    {
        std::uint32_t xm = (x0+x1) >> 1;            // 4. mid point
        if (vec[x0] < vec[xm])
        {
            // 1st half is monotonic                // 5. iterative update, with 2 cases
            if (contains_target(vec,x0,xm,target)])  x1 = xm;
            else                                    x0 = xm;
        }
        else
        {
            // 2nd half is monotonic                // 5. iterative update, with 2 cases
            if (contains_target(vec,xm,x1,target))  x0 = xm;
            else                                    x1 = xm;
        }
    }

    if (vec[x0] == target) return std::make_optional(x0);   // 6. final output
    if (vec[x1] == target) return std::make_optional(x1);
    return std::nullopt;
}

bool contains_target(const vector<int32_t>& vec, uint32_t x0, uint32_t x1, int32_t target)
{
    return vec[x0] <= target && target <= vec[x1];
}
```





3. Vector and string

Question 1 – First revisited element

Find the first character (or unsigned byte) that appears more than once in a vector, such that the set of possible elements is smaller than the size of vector. Solve it in *O(N)* time and *O(1)* space. Please refer to *Volant trading* and *codility.doc*.

```
int multi_instance(std::vector<int> &vec)
{
    for(int x:vec)
    {
        // 3 core variables : value, index and visited
        int value = abs(x);
        int index = value - 1;
        bool visited = (vec[index]<0);
        if  (visited) return value;
        else vec[index] = -vec[index];
    }
}
```

`

Question 2 - Passing Cars

Array *A* contains only 0s and 1s, with 0 represents a car traveling east, while 1 represents car traveling west. Implement function to count passing cars with O(N) time and O(1) space.

```
int solution(vector<int> &A)
{
     int num_zero = 0;
     int num_pass = 0;
     for(auto a : A)
     {
          if (a==0) ++num_zero;
          else num_pass += num_zero;
     }
     return num_pass;
}
```

Question 3 - Binary gap

Binary gap of a positive integer *N* is the size of longest sequence of consecutive zeros that is surrounded by ones at both ends in the binary representation of *N*. For example, number *9* has binary representation *1001* and contains a binary gap of length *2*. Number *529* has binary representation *1000010001* and contains two binary gaps, which have length *4* and *3* respectively. Number 20 has binary representation *10100* and contains one binary gap of length *1*. Write a function to count the length of gap.

```
int solution(int N)
{
     int count = 0;
     int max_count = 0;
     bool started_counting = false;

     while(N>0)
     {
          bool is_one = N%2;
          if (!started_counting)
          {
               if (is_one) started_counting = true;
          }
          else
          {    if (!is_one) ++count;
               else
               {
                    if (max_count < count) max_count = count;
                    count = 0;
               }
          }
          N = N/2;
     }
     return max_count;
}
```

Question 4 - Minimium average of subarray

A non-empty zero-indexed array *A* consisting of *N* integers is given. A pair of integers *(P, Q)* such that $0 \le P < Q < N$, is called a slice of array *A* (notice that the slice contains at least 2 elements). The average of a slice *(P,Q)* is the sum of *A[P]+A[P+1]+ ...+A[Q]* divided by the length of the slice. To be precise, the average equals *(A[P]+A[P+1]+...+A[Q])/(Q−P+1)*, the target is to find the starting position of a slice whose average is minimal. Worst case time complexity is *O(N)*.

```
// The solution is either moving average with window size 2 or 3 (i.e. MA2 or MA3).
// The reason is that extremum moving average must be MA1, now window size of 1 is not allowed.
```

Can we prove it? First of all, we have :

$$\begin{aligned} min(x_0, x_1) &\le (x_0+x_1)/2 \le max(x_0, x_1) && (1) \\ min(x_0, x_1) &\le w_0x_0+w_1x_1 \le max(x_0, x_1) \quad where\ w_0+w_1=1 && (2) \end{aligned}$$

Please note that *(2)* is a generic version of *(1)*, it represents the fact that optimal point of linear programming is always on vertex. As subarray size (let it be M) cannot be *1*, hence the next possible case is $M = 2$. Yet, do we need to ***consider M = 3*** as well? Yes, because there exists $[y_0, y_1, y_2]$ fulfilling *(3)*:

$$(y_0+y_1+y_2)/3 \le (x_0+x_1)/2 \le min[(y_0+y_1)/2, (y_1+y_2)/2] \qquad (3)$$

where $[x_0, x_1]$ is subarray with minimum average when $M = 2$, while $[y_0, y_1, y_2]$ is next subarray we are going to find for $M = 3$. Like :

$$\begin{aligned} [y_0, y_1, y_2] &= [0, 1, 0] \\ [x_0, x_1] &= [0.4, 0.4] \\ A[n] &= 1000 \text{ for all other } n \in [0,N) \end{aligned}$$

When we ***consider M = 4***, we can never find $[z_0, z_1, z_2, z_3]$ fulfilling both *(4a)* and *(4b)* :

$$(z_0+z_1+z_2+z_3)/4 \le (x_0+x_1)/2 \le min[(z_0+z_1)/2, (z_1+z_2)/2, (z_2+z_3)/2] \qquad (4a)$$

and

$$(z_0+z_1+z_2+z_3)/4 \le (y_0+y_1+y_2)/3 \le min[(z_0+z_1+z_2)/3, (z_1+z_2+z_3)/3] \qquad (4b)$$

Condition *(4)* can never be fulfilled, because :

$$\begin{aligned} (z_0+z_1+z_2+z_3)/4 &= [(z_0+z_1)/2 + (z_2+z_3)/2] / 2 \\ &\ge min[(z_0+z_1)/2, (z_2+z_3)/2] && by\ applying\ (2) \\ &\ge min[(z_0+z_1)/2, (z_1+z_2)/2, (z_2+z_3)/2] \end{aligned}$$

When we ***consider M = 5***, we can never find $[u_0, u_1, u_2, u_3, u_4]$ fulfilling *(5)* :

$$\begin{aligned} (w_0+w_1+w_2+w_3+w_4)/5 \le\ & min[(w_0+w_1)/2, (w_1+w_2)/2, (w_2+w_3)/2, (w_3+w_4)/2, \\ & (w_0+w_1+w_2)/3, (w_1+w_2+w_3)/3, (w_2+w_3+w_4)/3 \\ & (w_0+w_1+w_2+w_3)/4, (w_1+w_2+w_3+w_4)/4] \qquad (5) \end{aligned}$$

Condition *(5)* can never be fulfilled, because :

$$\begin{aligned} (w_0+w_1+w_2+w_3+w_4)/5 &= 0.4 \times (w_0+w_1)/2 + 0.6 \times (w_2+w_3+w_4)/3 \\ &\ge min[(w_0+w_1)/2, (w_2+w_3+w_4)/3] && by\ applying\ (2) \\ &\ge min[(w_0+w_1)/2, (w_1+w_2)/2, (w_2+w_3)/2, (w_3+w_4)/2, \\ &\qquad (w_0+w_1+w_2)/3, (w_1+w_2+w_3)/3, (w_2+w_3+w_4)/3 \\ &\qquad (w_0+w_1+w_2+w_3)/4, (w_1+w_2+w_3+w_4)/4] \end{aligned}$$

Question 5 Reverse words in a sentence inplace, i.e. from `“this is a pen”` to `“pen a is this”`. Do it inplace. We need 2 scans :

- reverse a string character-wise
- reverse a word character-wise, do it for each word

Question 6 Permutation of a set of characters using recursion.

```
std::vector<std::string> permutation(std::set<char> chars)
{
    std::vector<std::string> output, temp;
    if (chars.empty()) return output;
    chars.erase(chars.begin());
    permutation(chars, temp);

    output.push_back(std::string(1, *(chars.begin())));
    for(const auto& s : temp)
    {
        output.push_back(s);
        for(unsigned n=0; n<=s.size(); ++n)
        {
            std::string s0 = s; s0.insert(n,1,c);
            output.push_back(s0);
        }
    }
    return output;
}
```

4. Stack and Queue

Question 1 Implement stack with singly linked list, implement queue with doubly linked list.
Question 2 Implement queue with two stacks and implement stack with two queues.

Idea for `my_queue` :

- always push to stack0
- always pop from stack1
- migrate items from stack0 to stack1 **only** when the latter is empty on popping

Idea for `my_stack` :

- only one queue is non-empty
- always push to non-empty queue
- always pop from non-empty queue **after** migrating all-but-except-the-last items to the emty queue

Provide 6 basic functions for each container : push, pop, front (or top), empty, size and clear.

```
template<typename T> class my_queue // This solution is verified in MSVS.
{
public:     void push(const T& x)           {                         stack0.push(x); }
            void pop()                      { move_from_0to1();       stack1.pop();   }
            T&   front() const              { move_from_0to1(); return stack1.top();  }
            bool empty() const              { return stack0.empty() && stack1.empty(); }
            auto size()  const              { return stack0.size()  +  stack1.size();  }
            void clear()                    {        stack0.clear();   stacl1.clear(); }

private:    void move_from_0to1() const // declare const, since called by front()
            {
                if (!stack1.empty()) return;
                while(!stack0.empty())
                {
                    stack1.push(stack0.top());
                    stack0.pop();
                }
            }

private:    mutable std::stack<T> stack0; // declare mutable, since called by move_from_0to1
            mutable std::stack<T> stack1; // declare mutable, since called by move_from_0to1
};

template<typename T> class my_stack // This solution is verified in MSVS.
{
public:     my_stack() : active0(true) {}

            void push(const T& x)           { if (active0) queue0.push(x);       else queue1.push(x);        }
            void pop()                      { if (active0) move_from_0to1();     else move_from_1to0();      }
            T&   top()   const              { if (active0) return queue0.back(); else return queue1.back();  }
            bool empty() const              { return queue0.empty() && queue1.empty();                       }
            auto size()  const              { return queue0.size()  +  queue1.size();                        }
            void clear()                    {        queue0.clear();   queue.clear();  active0 = true;       }

private:    void move_from_0to1()
            {
                while(!queue0.empty())
                {
                    if (queue0.size()>1) queue1.push(queue0.front());
                    queue0.pop();
                }
                active0 = false;
            }

            void move_from_1to0()
            {
                while(!queue1.empty())
                {
                    if (queue1.size()>1) queue0.push(queue1.front());
                    queue1.pop();
                }
                active0 = true;
            }

private:    bool active0;
            std::queue<T> queue0;
            std::queue<T> queue1;
};
```

Question 3 Implement multiple stacks with one array. This question can be extended to lists-on-array (a little more complicated).

Solution

We need 3 classes : `node`, `stack` and `obj_pool`

- each `node` has a ***link*** to next `node`
- each `stack` has a ***link*** to its root, it offers 6 functions : push, pop, top, empty, size and clear *(like question2)*
- `obj_pool` has a ***link*** to unused list, it offers 2 functions : request and release
- `obj_pool` does not keep check of the `stack`, it doesn't even know the number of prevailing `stacks`
- `node<T>*` is used instead of `std::int16_t` as next node pointer, it offers better speed for `iterator<T>::operator++()`

  i.e. `node_ptr = node_ptr->next` is faster than `node_idx = pool[node_idx].next`

```
template<typename T> struct node
{
    T value;
    node<T>* next = nullptr;
};

template<typename T> struct stack
{
    stack(obj_pool<T>& _pool) : pool(_pool), root(nullptr), size(0) {}

    template<typename... ARGS> void push(ARGS&&... args)
    {
        auto new_node = pool.request(std::forward<ARGS>(args)...); if (!temp) return; // do nothing on no memory
        new_node->next = root;
        root = new_node;
        ++size;
    }

    void pop()
    {
        auto del_node = root; if (!temp) return; // do nothing on popping empty stack
        root = root->next;
        pool.release(del_node);
        --size;
    }

    const T& top() const { return  root -> value;    }
    bool empty()   const { return (root == nullptr); }
    auto size()    const { return size;              }
    void clear()         { while (!empty()) pop();   }

private:
    obj_pool<T>& pool; // dependency injection
    node<T>* root;
    std::size_t size;
};
```

The implementation steps of `stack::push` is similar to that of `obj_pool::release`. The implementation steps of `stack::pop` is similar to that of `obj_pool::request`. Since `release` is pushing to unused list, while `request` is popping from unused list.

```
template<typename T> struct obj_pool
{
    obj_pool(std::uint16_t size) : pool(size), unused_root(&pool[0])
    {
        for(std::uint16_t n=0; n!=size-1; ++n) pool[n].next = &pool[n+1];
        pool[size-1].next = nullptr;
    }

    template<typename... ARGS> node<T>* request(ARGS&&... args)
    {
        if (!unused_root) return nullptr; // no memory

        auto new_node = unused_root;                               // new_node for stack == del_node for unused_list
        unused_root   = unused_root->next;
        new (&(new_node->value)) T{std::forward<ARGS>(args)...};
        return new_node;
    }

    void release(node<T>* del_node)
    {
        if (!del_node) return; // popping empty stack

        del_node->value.~T();                                      // del_node for stack == new_node for unused_list
        del_node->next  = unused_root;
        unused_root     = del_node;
    }

private:
    std::vector<node<T>> pool;
    node<T>* unused_root;
};

obj_pool<int> pool(100);
stack<int> s0(pool); s0.push(10); s0.push(11); s0.push(12);
stack<int> s1(pool); s1.push(20); s1.push(21);
stack<int> s2(pool); s2.push(30);
```

Question 4 Design a stack with member function that returns minimum of current values.

Method 1 : Auxiliary stack
Whenever a new value is pushed, push current minimum into auxiliary stack as well.

```
template<typename T> class my_stack // This solution is verified in MSVS.
{
public:       void push(const T& x)
              {
                  values.push(x);
                  if (minimum.empty())           minimum.push(x);
                  else if (x < minimum.top())    minimum.push(x);
                  else                           minimum.push(minimum.top());
              }
              void pop()              { values.pop(); minimum.pop(); }
              const T& min() const    { return minimum.top();   }
              const T& top() const    { return values.top();    }
              bool empty() const      { return values.empty();  }

private:      std::stack<T> values;
              std::stack<T> minimum;
};
```

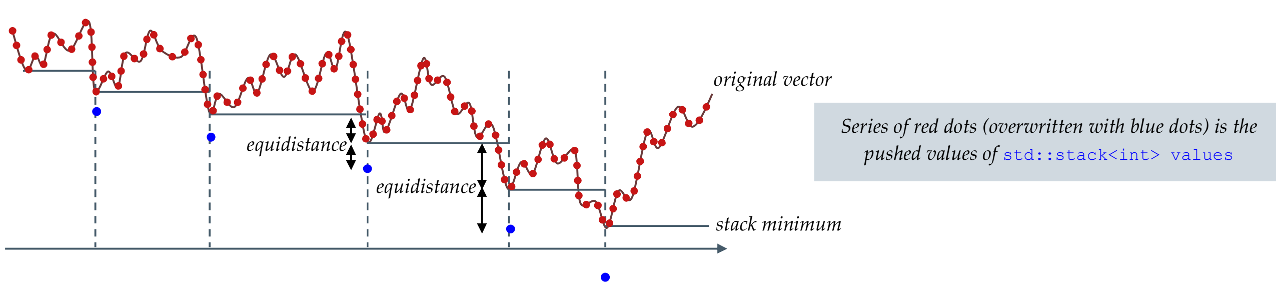
Method 2 : No auxiliary stack
This trick works for integer-stack only. There is no auxiliary containers, but an auxiliary variable `minimum` storing current minimum.

- as we add integer `x` into the stack, for most of the time, `minimum` is unchanged
- `values.top()` should be greater than or equal to `minimum`
- `values.top()` smaller than `minimum` can be used as a boolean to indicate an event, such as a change in `minimum`
- when adding an integer `x` smaller than `minimum`, doing both `values.push(x)` and `minimum=x` is a waste (it duplicates) …
- instead we update `minimum=x` while pushing into `values` a number which is :

- smaller than `x` and
- allow us to retrieve the original `minimum`

- to do so, `values.push(x – (minimum_old-minimum_new))` where `minimum_new = x`, hence we have `values.push(2x-minimum_old)`
- conversely, as we pop, if we find that `values.top() < minimum`, then we have to update `minimum` by … see code below

```
class int_stack // This solution is verified in MSVS.
{
public:       int_stack() : minimum(1000) {} // std::numeric_limits<int>::max()
              void push(int x)
              {
                  if (x < minimum)    { values.push(2*x - minimum); minimum = x; }
                  else                  values.push(x);
              }
              void pop()              { if (minimum > values.top()) minimum = 2*minimum - values.top(); values.pop(); }
              int  top() const        { if (minimum > values.top())      return minimum;    else return values.top(); }
              int  min() const        { return minimum; }
              bool empty() const      { return values.empty(); }

private:      std::stack<int> values;
              int minimum;
};
```



5. List

Question 1 Implement node insertion / deletion in singly linked list and doubly linked list. For simplicity, both singly and doubly linked list do not have extra end node. That is, for singly linked list, the last node has its next pointer pointing to null. For insertion, we would like to offer two versions, insert-before (`STL` adopts this convention) and insert-after, please update `head_node` & `tail_node` appropriately, Besides as there is no end node, we need to provide `push_back` for insertion to end using `insert_before`, similarly we do provide `push_front` as well. First of all, singly list insertion :

```
template<typename T, typename...ARGS> void singly_list<T>::push_front(ARGS...&& args)
{
    // *** Core part *** //
    auto new_node   =  new node<T>(std::forward<ARGS>(args)...);
     new_node->next = head_node;
    head_node       =  new_node;
}

template<typename T, typename...ARGS> void singly_list<T>::push_back(ARGS...&& args)
{
    // *** Delegation *** //
    if (!head_node) { push_front(std::forward<ARGS(args)...>); return; }

    // *** Find last node *** //
    auto last_node = head_node;
    while(last_node->next) last_node = last_node->next;

    // *** Core part *** //
    auto new_node   = new node<T>(std::forward<ARGS>(args)...);
     new_node->next = nullptr;
    last_node->next = new_node;
}

template<typename T, typename...ARGS> void singly_list<T>::insert_before(node<T>* this_node, ARGS...&& args)
{
    // *** Delegation *** //
    if (!this_node) { push_back(std::forward<ARGS(args)...>); return; }

    // *** Find prev node *** //
    auto prev_node = head_node;
    while(prev_node->next != this_node) prev_node = prev_node->next;

    // *** Core part *** //
    auto new_node   =  new node<T>(std::forward<ARGS>(args)...);
     new_node->next = this_node;
    prev_node->next =  new_node;
}

template<typename T, typename...ARGS> void singly_list<T>::insert_after(node<T>* this_node, ARGS...&& args)
{
    // *** Core part *** //
    auto new_node   =  new node<T>(std::forward<ARGS>(args)...);
     new_node->next = this_node->next;
    this_node->next =  new_node;
}
```

Next, doubly list insertion :

```
template<typename T, typename...ARGS> void doubly_list<T>::push_front(ARGS...&& args)
{
    auto new_node = new node<T>(std::forward<ARGS>(args)...);
    if (!head_node) // then tail_node must be nullptr as well
    {
         new_node->next = nullptr;
         new_node->prev = nullptr;
        head_node = new_node;
        tail_node = new_node;
    }
    else
    {
         new_node->next = head_node;
         new_node->prev = nullptr;
        head_node->prev = new_node;
        head_node       = new_node;
    }
}

template<typename T, typename...ARGS> void doubly_list<T>::push_back(ARGS...&& args)
{
    if (!head_node) { push_front(std::forward<ARGS(args)...>); return; }

    auto new_node = new node<T>(std::forward<ARGS>(args)...);
     new_node->next = nullptr;
     new_node->prev = tail_node;
    tail_node->next = new_node;
    tail_node       = new_node;
}
```

```
template<typename T, typename...ARGS> void doubly_list<T>::insert_before(node<T>* this_node, ARGS...&& args)
{
    if (!this_node) return;
//  if (!head_node || !tail_node) no need, if this_node is in the list, head_node / tail_node cannot be nullptr

    auto  new_node  = new node<T>(value...);
    auto prev_node  = this_node->prev;
     new_node->next = this_node;
     new_node->prev = prev_node;
    prev_node->next =  new_node;
    this_node->prev =  new_node;
}

template<typename T, typename...ARGS> void doubly_list<T>::insert_before(node<T>* this_node, ARGS...&& args)
{
    if (!this_node) return;
//  if (!head_node || !tail_node) no need, if this_node is in the list, head_node / tail_node cannot be nullptr

    auto  new_node  =  new node<T>(value...);
    auto next_node  = this_node->next;
     new_node->next = next_node;
     new_node->prev = this_node;
    this_node->next =  new_node;
    next_node->prev =  new_node;
}
```

Deletion in singly linked list is complicated. Two problems with O(1) solution :

- it involves deep copy of node value and
- it may result in dangling point, deleting `next_node` which may be dereferenced by other objects

```
template<typename T> void singly_list<T>::erase(node<T>* this_node)
{
    // skip update of head_node for simplicity
    auto next_node = this_node->next;
    this_node->next = next_node->next;
    this_node->value = next_node->value;          // Problem 1 : deep copy of value, may be slow
    delete next_node;                             // Problem 2 : iter pointing to next_node becomes invalid
}

template<typename T> void doubly_list<T>::erase(node<T>* this_node)
{
    // skip update of head_node & tail_node for simplicity
    auto prev_node = this_node->prev;
    auto next_node = this_node->next;
    prev_node->next = next_node;
    next_node->prev = prev_node;
    delete this_node;
}
```

List reversal in singly linked list and doubly linked list involve 3 steps in each iteration inside :

```
template<typename T> void singly_list<T>::reverse()
{
    node<T>* prev_node = nullptr;
    node<T>* this_node = head_node;
    while(this_node != nullptr)
    {
        node<T>* next_node = this_node->next;
        this_node->next = prev_node;
        prev_node = this_node;
        this_node = next_node;
    }
    head_node = prev_node;
}

template<typename T> void doubly_list<T>::reverse()
{
    node<T>* this_node = head_node;
    while(this_node != nullptr)
    {
        node<T>* next_node = this_node->next;
        std::swap(this_node->prev, this_node->next);
        this_node = next_node;
    }
    std::swap(head_node, tail_node);
}
```

Names of different nodes :

```
* head_node, tail_node
* this_node, next_node, prev_node
*  new_node,  del_node
```

Question 2 Find the n[th] node from the end of a singly linked list in single scan.

```
template<typename T> node<T>* singly_list<T>::nth_node_from_end(int N)
{
    node<T>* node0 = head_node;
    node<T>* node1 = head_node;
    for(int n=0; n!=N; ++n) node1 = node1->next;
    while(node1 != nullptr)
    {
        node0 = node0->next;
        node1 = node1->next;
    }
    // when N=0, return nullptr
    // when N=1, return rbegin
    // when N=list.size(), return begin
    return node0;
}
```

Question 3 How to detect if two singly linked lists intersect? If two singly linked lists intersect, they must share the same end node, hence the solution is simple : trace both lists until reaching their end nodes, if they are identical, then the two lists must be partially overlapping. How to detect if two doubly linked list intersect? If they do, they must be identical to each other (as list does not have branches, if a list has branches, it becomes a tree).

Question 4 How to detect loop in singly linked list? Please read Volant Trading interview 2. The implementation is shown below, it detects if there exists loop, measures loop size and finds loop entry point. How to detect loop in doubly linked list? If doubly linked list has a loop, the list must become a circular linked list. *Remark : Don't forget to use `**` for outputting node pointer.*

```
template<typename T> bool singly_list<T>::detect_loop(node<T>** meet_node)
{
    node<T>* slow_node = head;
    node<T>* fast_node = head;
    while(true)
    {
        if (slow_node != nullptr) slow_node = slow_node->next; else return false; // slow node jumps 1 step
        if (fast_node != nullptr) fast_node = fast_node->next; else return false; // fast node jumps 2 steps
        if (fast_node != nullptr) fast_node = fast_node->next; else return false;
        if (slow_node == fast_ptr) { *meet_node = slow_node; return true; }
    }
}

template<typename T> int singly_list<T>::loop_length(node<T>* meet_node)
{
    node<T>* this_node = meet_node; int size  = 0;
                                    this_node = this_node->next; ++size;
    while(this_node != meet_node) { this_node = this_node->next; ++size; }
    return size;
}

template<typename T> bool void singly_list<T>::loop_entry(node<T>* meet_node, node<T>** entry_node)
{
    node<T>* node0 = head;
    node<T>* node1 = meet_node;                                                  0 1 2 3 x x e x x x x
    while(true)                                                                              n       x
    {                                                                                        x       x
        if (node0 != nullptr) node0 = node0->next; else return false;                        x       x
        if (node1 != nullptr) node1 = node1->next; else return false;                        x       x
        if (node0 == node1) { *entry_node = node0; return true; }                            x x m x x
    }
}
```

How does `loop_entry` work? Nodes in the list are indiced. Let node *0*, node *e*, node *m* and node *n* be `head`, `entry_node`, `meet_node` and the last-node-in-the-loop respectively. We have :

- distance travelled by `slow_node` $= m$
- distance travelled by `fast_node` = *distance of whole list* + *distance from (e-1) to m* $= n + m - (e-1)$, which is double of `slow_node`
- hence we have : $2m = n + m - (e-1)$, or equivalently $m = n - e + 1$, and eventually $e = (n+1) - m$
- hence distance from `head` to `entry_node` equals to distance from `meet_node` to next round's `entry_node`

6. Binary search tree

Definitions

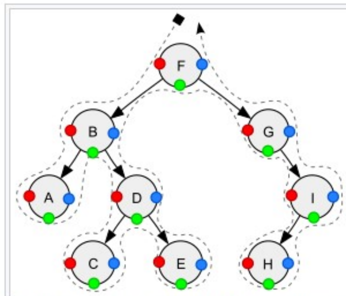
1. Tree can be defined in two ways :
- a tree is a root node plus multiple branches, each connects to the root node of non-overlapping subtrees,
- a tree is a graph in which, there exists one and only one path between any two nodes
- which is the one that routes through their common ancester.
2. Binary tree is a tree in which, each node has at most two children.
3. Binary search tree is a binary tree in which :

```
max_key(subtree_rooted_on(this_node.lhs_child)) ≤ this_node.key ≤ min_key(subtree_rooted_on(this_node.rhs_child))
```

4. Balanced tree is a binary tree with maximum depth and minimum depth of leaves differ by at most one
- we define balance factor of `node` :

`depth(node) = 1`	if `node` is a leaf
`depth(node) = max(depth(node.lhs_child),depth(node.rhs_child)) + 1`	if `node` is not a leaf
`BF(node)    =    depth(node.lhs_child)-depth(node.rhs_child)`	hence `BF∈[-1,-0,+1]` for balanced tree

5. Each node has **one key**, cutting the key-space into two halves (in `set` and `map`), can be generalized to **multiple keys** (in Btree).
6. Each node has either **link** to children, or **link** to parent, or both. Link to parent can be implemented as **hashmap**.
7. null check is done for `this_node` for function `find` and `insert`, there are 2 cases :
   `this_node == nullptr`
   or `this_node != nullptr`
   null check is done for `this_node->lhs` and `this_node->rhs` in other questions, there are 4 cases :
   `this_node->lhs == nullptr` and `this_node->rhs == nullptr`
   or `this_node->lhs == nullptr` and `this_node->rhs != nullptr`
   or `this_node->lhs != nullptr` and `this_node->rhs == nullptr`
   or `this_node->lhs != nullptr` and `this_node->rhs != nullptr`



Depth-first traversal (dotted path) of a binary tree:
Pre-order (node visited at position red ●):
F, B, A, D, C, E, G, I, H;
In-order (node visited at position green ●):
A, B, C, D, E, F, G, H, I;
Post-order (node visited at position blue ●):
A, C, E, D, B, H, I, G, F.

Offering 6 functions

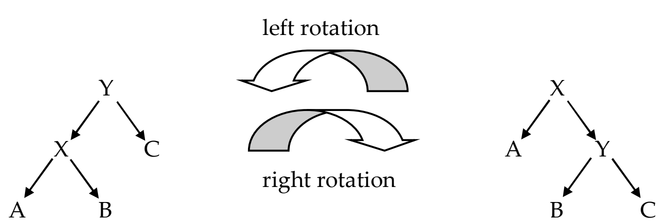
- depth / balance factor
- find
- insert / delete
- traverse (BFS / DFS-pre-order / DFS-in-order / DFS-post-order)
- rotate
- rebalance (which invokes rotate)

Traversal

- breadth first search (BFS) works like region growing using queue, it visits nodes layer by layer
- depth first search (DFS) works like region growing using stack, it visits node in order (if this is a search tree)
- DFS can further be classified as pre-order, in-order and post-order, ***BFS does not***

	recursive implement	iterative implement
BFS	no	region growing with a queue
DFS-pre-order	yes, trivial	region growing with a stack
DFS-in-order	yes, trivial	need a stack in iterator (complicated)
DFS-post-order	yes, trivial	need a stack in iterator (complicated)

Rotation is illustrated as :



Balance factor of A, B & C are unchanged by rotation. Consider a unbalanced subtree with root Y, where `BF(Y)=+2`
case 1 : `depth(A) = depth(B)   = depth(C)+1, BF(X)=0`   by right rotate, becomes balanced subtree with root X, where `BF(X)=-1`
case 2 : `depth(A) = depth(B)+1 = depth(C)+1, BF(X)=+1`   by right rotate, becomes balanced subtree with root X, where `BF(X)=0`
case 3 : `depth(A)+1 = depth(B) = depth(C)+1, BF(X)=-1`   by right rotate, still unbalanced subtree with root X, where `BF(X)=-2`

Question 1 Implement the following functions for *AVL* tree : depth, insert, find, tranverse, rotate and rebalance.

```
template<typename T> struct node
{
    T value;
    node<T>* lhs;
    node<T>* rhs;
};

template<typename T> class AVL_tree
{
    auto depth_and_balance_factor(const node<T>* this_node)
    {
        if (this_node == nullptr) return 0;
        return std::make_pair(1 + std::max(depth(this_node->lhs), depth(this_node->rhs)),
                                         depth(this_node->lhs) - depth(this_node->rhs));
    }

    node<T>* find(const node<T>* this_node, const T& value)
    {
        if (this_node == nullptr)           return nullptr;
        if (this_node->value == value)      return this_node;
        if (this_node->value <  value)      return find(this_node->rhs, value);
        else                                return find(this_node->lhs, value);
    }

    node<T>* insert(node<T>* this_node, const T& value) // return newly created node
    {
        if (this_node == nullptr)       {   this_node = new node<T>(value, nullptr, nulltr); return this_node; }
        if (this_node->value == value)      return this_node; // suppose this is std::set, not std::multiset
        if (this_node->value <  value)      return insert(this_node.rhs, value);
        else                                return insert(this_node.lhs, value);
    }
};
```

Traversal

Recursive implementation is trivial, as it makes use of ***call stack*** for region growing.

```
void DFS_recursion(node<T>* this_node, std::function<void(const T&)> fct, int mode)
{
    if (this_node == nullptr)               return;
    if (mode == pre_order)              {   fct(this_node->value);
                                            DFS_recursion(this_node->lhs, fct);
                                            DFS_recursion(this_node->rhs, fct);   }
    if (mode == in_order)               {   DFS_recursion(this_node->lhs, fct);
                                            fct(this_node ->value);
                                            DFS_recursion(this_node->rhs, fct);   }
    if (mode == post_order)             {   DFS_recursion(this_node->lhs, fct);
                                            DFS_recursion(this_node->rhs, fct);
                                            fct(this_node ->value);               }
}
```

Iterative implementation for <u>DFS-pre-order</u> and <u>BFS</u> are also trivial, they use an explicit ***stack/queue*** for region growing. The idea is to put all ***visited-but-not-yet-processed*** neighbouring nodes into a stack or a queue for future process. Never push `nullptr` in stack.

```
void BFS_iterative(node<T>* this_node, std::function<void(const T&)> fct) // no order option
{
    std::queue<node<T>*> q;
    if (this_node) q.push(this_node);

    while (!q.empty())
    {
        this_node = q.front(); q.pop();
        fct(this_node->value);
        if (this_node->lhs) q.push(this_node->lhs);
        if (this_node->rhs) q.push(this_node->rhs);
    }
}

void DFS_pre_order_iterative(node<T>* this_node, std::function<void(const T&)> fct) // for pre-order only
{
    std::stack<node<T>*> s;
    if (this_node) s.push(this_node);

    while (!s.empty())
    {
        this_node = s.top(); s.pop();
        fct(this_node->value);                    // line 1
        if (this_node->lhs) s.push(this_node->rhs); // line 2
        if (this_node->rhs) s.push(this_node->lhs); // line 3
    }   // if move line 1 beyond line 2&3, it is still pre-order (not in-order nor post-order)
}       // if swap line 2&3, it becomes descending pre-order (not ascending post-order)
```

Iterative implementation for DFS-in-order and DFS-post-order are complicated. We will walk through DFS-in-order only.

The idea is :

- `this_node` is the node under consideration, it *may be* `nullptr`, and we are traversing to LHS as deep as possible `remark 1`
- on reaching leaf `this_node==nullptr`, there are two cases : `remark 2`
  - `this_node` is the LHS child of its parent, then go to its parent and process it, finally go to its RHS sibling
  - `this_node` is the RHS child of its parent, then go to the next-deepest node having its *LHS subtree* visited, but not its *RHS subtree*
- in other words, we need to cache all *ancestors* of `this_node` in stack, with *LHS subtree* visited, but not its *RHS subtree*
- rewrite `DFS_pre_order_iterative` first :

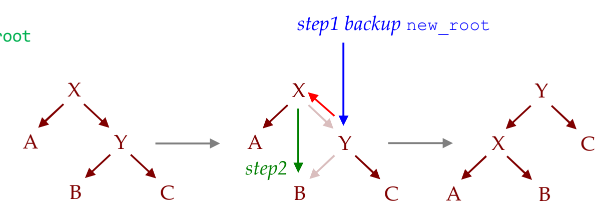
```
void DFS_pre_order_iterative_2(node<T>* this_node, std::function<void(const T&)> fct) // for pre-order only
{
    std::stack<node<T>*> s;
    while (!s.empty() || this_node)
    {
        if (this_node != nullptr)
        {
            fct(this_node->value);  // line 1
            s.push(this_node->rhs); // line 2
            s.push(this_node->lhs); // line 3
        }
        this_node = s.top(); s.pop();
    }
}

void DFS_in_order_iterative(node<T>* this_node, std::function<void(const T&)> fct)
{
    std::stack<node<T>*> s;
    while (!s.empty() || this_node)
    {
        if (this_node != nullptr) // implementation of remark 1
        {
            s.push(this_node);
            this_node = this_node->lhs;
        }
        else // implementation of remark 2
        {
            this_node = s.top(); s.pop();
            fct(this_node->value);
            this_node = this_node->rhs;
        }
    }
}
```

Is this an implementation of `std::set<T>::iterator`*?*
Yes, we need to add `std::stack<node<T>*>` *inside iterator !!!*
Search keyword `binary search tree iterator` *in web.*

Rotation

Rotation can be illustrated as :

```
node<T>* lhs_rotate(node<T>* this_node) // return new root
{
    if (this_node      == nullptr) return;
    if (this_node->rhs == nullptr) return;

    node<T>* new_root = this_node->rhs;
    this_node->rhs = new_root->lhs;
    new_root->lhs = this_node;
    return new_root;
}
```




```
node<T>* rhs_rotate(node<T>* this_node) // return new root
{
    if (this_node      == nullptr) return;
    if (this_node->lhs == nullptr) return;

    node<T>* new_root = this_node->lhs;
    this_node->lhs = new_root->rhs;
    new_root->rhs = this_node;
    return new_root;
}
```

Rebalance – naive approach

```
node<T>* balance(node<T>* this_node) // return new root, in O(N) time
{
    std::vector vec;
    vec.reserve(this->size());
    fill_array_by_DFS_inorder_traversal(vec); // elements in vec must be sorted
    return construct_BST_from_vec(vec);
}
```

Rebalance on insertion
The algorithm for rebalance on deletion is different.

When inserting a new node, that makes a balanced tree imbalanced, then :

- the new node must be a leaf which makes `BF(this_node->parent)` being `+1` or `-1`
- which in turn makes `BF(this_node->parent->parent)` being `+2` or `-2`
- which in turn makes `BF(this_node->parent->parent->parent)` being `+2` or `-2` (can switch from `+2` to `-2`, vice versa, like zig zag path)
- the propagation of `±2` balance factor continues until either : it reaches the root or it encounters the first `±1` balance factor

Given node `X` as the first node (nearest to leaf) having `BF=±2`, node `Y` as the bigger child of `X` having `BF=±1`, here are the 4 cases :

- `BF(X)=-2` and `BF(Y)=-1` balanced by left rotation at node `X`
- `BF(X)=-2` and `BF(Y)=+1` balanced by right rotation at node `Y`, followed by left rotation at node `X`
- `BF(X)=+2` and `BF(Y)=+1` balanced by right rotation at node `X` *(mirror image of the 1st case)*
- `BF(X)=+2` and `BF(Y)=-1` balanced by left rotation at node `Y`, followed by right rotation at node `X` *(mirror image of the 2nd case)*

The rebalancing at node `X` will propagate the updated depth to its parents, which eventually make the whole tree balanced.

case 1

```
                 X BF(X)=-2
        ┌────────┴────────┐
        Z                 Y BF(Y)=0/-1
  depth(Z)=N        ┌─────┴─────┐
                    Y0          Y1
             depth(Y0)=N+1 & depth(Y1)=N+1
          or depth(Y0)=N   & depth(Y1)=N+1

-> left rotate at X

                       Y BF(Y)=+1/0
              ┌────────┴────────┐
   BF(X)=-1/0 X                 Y1
        ┌─────┴─────┐          depth(Y1)=N+1
        Z           Y0      or depth(Y1)=N+1
   depth(Z)=N    depth(Y0)=N+1
              or depth(Y0)=N
```

case 2

```
                 X BF(X)=-2
        ┌────────┴────────┐
        Z                 Y BF(Y)=+1
  depth(Z)=N        ┌─────┴─────┐
                    Y0          Y1
                ┌───┴───┐       depth(Y1)=N
                Y00     Y01
                depth(Y00,Y01)={N,N}
             or depth(Y00,Y01)={N,N-1}
             or depth(Y00,Y01)={N-1,N}

-> right rotate at Y
->  left rotate at X

              X
     ┌────────┴────────┐
     Z                 Y0
                  ┌────┴────┐
                  Y00       Y
                        ┌───┴───┐
                        Y01     Y1

                       Y0 BF(Y0)=0/+1/+1
              ┌────────┴────────┐
 BF(X)=0/0/+1 X                 Y BF(Y)=0/-1/0
        ┌─────┴─────┐     ┌─────┴─────┐
        Z           Y00   Y01         Y1
  depth(Z)=N     depth(Y00,Y01)={N,N}     depth(Y1)=N
              or depth(Y00,Y01)={N,N-1}
              or depth(Y00,Y01)={N-1,N}
```

case 3

```
                    X BF(X)=+2
           ┌────────┴────────┐
 BF(Y)=0/+1 Y                Z
      ┌─────┴─────┐          depth(Z)=N
      Y0          Y1
   depth(Y0)=N+1 & depth(Y1)=N+1
or depth(Y0)=N+1 & depth(Y1)=N

-> right rotate at X

                 Y BF(Y)=-1/0
        ┌────────┴────────┐
        Y0                X BF(X)=+1/0
   depth(Y0)=N+1    ┌─────┴─────┐
or depth(Y0)=N+1    Y1          Z
               depth(Y1)=N+1    depth(Z)=N
            or depth(Y1)=N
```

case 4

```
                    X BF(X)=+2
           ┌────────┴────────┐
  BF(Y)=-1 Y                 Z
      ┌─────┴─────┐          depth(Z)=N
      Y0          Y1
depth(Y1)=N   ┌───┴───┐
              Y10     Y11
              depth(Y10,Y11)={N,N}
           or depth(Y10,Y11)={N-1,N}
           or depth(Y10,Y11)={N,N-1}

->  left rotate at Y
-> right rotate at X

                    X
           ┌────────┴────────┐
           Y1                Z
      ┌────┴────┐
      Y         Y11
  ┌───┴───┐
  Y0      Y10

                       Y1 BF(Y1)=0/+1/+1
              ┌────────┴────────┐
 BF(Y)=0/+1/0 Y                 X BF(X)=0/0/-1
        ┌─────┴─────┐     ┌─────┴─────┐
        Y0          Y10   Y11         Z
  depth(Y0)=N    depth(Y10,Y11)={N,N}     depth(Z)=N
              or depth(Y10,Y11)={N-1,N}
              or depth(Y10,Y11)={N,N-1}
```

Question 2 Given a vector, verify if it follows a post-order traversal sequence of a binary search tree.

```
template<typename ITER> bool is_post_order(ITER begin, ITER end)
{
     if (begin == end)  return true;      ITER last = end-1;
     if (begin == last) return true;      ITER mid  = last; // if all elements are smaller than last,
                                                            // then all elements are in LHS subtree.

     for(ITER i=begin; i!=last; ++i) { if (*i>*last) { mid = i; break; }}
     for(ITER i=mid;   i!=last; ++i) { if (*i<*last)   return false;    }
     return is_post_order(begin, mid) && is_post_order(mid, last);
}
```

Question 3 Given a binary tree, verify if it is sorted.

```
template<typename T>
bool is_search_tree (   node<T>* this_node,
                        const T& lower = std::numeric_limits<T>::min,
                        const T& upper = std::numeric_limits<T>::max)
{
    if (!this_node) return true;

    if (this_node->lhs)  { if (!is_search_tree(this_node->lhs, lower, this_node->value)) return false; }
    else                 { if (this_node->value < lower)                                 return false; }
    if (this_node->rhs)  { if (!is_search_tree(this_node->rhs, this_node->value, upper)) return false; }
    else                 { if (this_node->value > upper)                                 return false; }
    return true;
}
```

Question 4 Given a sorted vector, build a balanced binary tree. This question comes from CASH-CTI interview.

```
template<typename ITER>
node<typename std::iterator_traits<ITER>::value_type>* build_binary_tree(ITER begin, ITER end)
{
     typedef typename std::iterator_traits<ITER>::value_type T;
     unsigned long size = std::distance(begin, end);
     if (size==0) return nullptr;

     unsigned long middle = size/2;
     node<T>* root = new node<T>(*(begin+middle));
     node->lhs = build_binary_tree(begin, begin+middle);
     node->rhs = build_binary_tree(begin+middle+1, end);
     return root;
}
```

The recursions in question 2,3,4 are easier as they return bool or just a root node. However the recursion in question 5 is the head or the tail of a doubly linked list.

Question 5 Convert binary tree into double-linklist inplace, using LHS pointer as previous pointer and RHS pointer as next pointer.

```
template<typename T>
void tree2dlist(const node<T>* this_node, node<T>** head_ptr, node<T>** tail_ptr)
{
    if (this_node->lhs)
    {
        node<T>* temp_node;
        tree2dlist(this_node->lhs, head_ptr, &temp_node);
        this_node->lhs = temp_node;
        temp_node->rhs = this_node
    }
    else // this_node->lhs = nullptr
    {
        *head_ptr = this_node;
    }

    if (this_node->rhs)
    {
        node<T>* temp_node;
        tree2dlist(this_node->rhs, &temp_node, tail_ptr);
        this_node->rhs = temp_node;
        temp_node->lhs = this_node;
     }
    else // this_node->rhs = nullptr
    {
        *tail_ptr = this_node;
    }
}
```

- for trees implementation, we check nullity for `this_node`
- for question 3 and 5, we check nullity for `this_node->lhs` and `this_node->rhs` instead

7. Heaptree / Btree / Skip list / Prefix tree (Trie)

7.1 Heaptree

(1) Binary heap is complete binary tree : all layers (except bottom layer) fully filled, bottom layer is filled from left to right
(2) Binary heap is partially sorted : node value is smaller than both of its children *(different ordering as compared to binary tree)*
(3) Binary heap is implemented as an array with …
(4) node *0* as top-most layer

- children of node *n* are node *2n+1* and node *2n+2* respectively
- ancestor of node *n* is node *(n-1)/2*

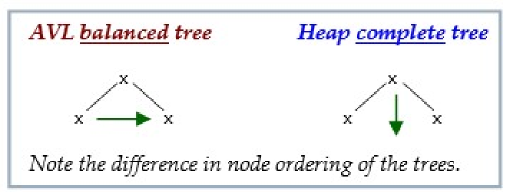
Pushing a new value is done by :

- insertion of the new value into bottom layer
- perform floating, i.e. swap it with its parent until binary heap order is maintained *(floating doesn't consider branch)*

Popping value is done by :

- removing the root, replacing it with the last element
- perform sinking, i.e. swap it with the smaller child until binary heap order is maintained *(sinking does consider branch)*

```
template<typename T> class heap
{
public:
    void     push(const T& x)   {    key.push_back(x);                          float_key();  }
    void     pop()              {    key.front()=key.back(); key.pop_back();    sink_key();   }
    const T& top()   const      {    return key.front();                                      }
    int      size()  const      {    return key.size ();                                      }
    bool     empty() const      {    return key.empty();                                      }
    void     clear()            {    key.clear();                                             }

private:
    void float_key()
    {
        int n = size()-1;
        while(n!=0)
        {
            int m = (n-1)/2;
            if (key[n] < key[m]) { key.swap(n, m); n = m; }
            else return;
        }
    }

    void sink_key()
    {
        int n = 0;
        while(true)
        {
            int m0 = n*2 + 1;   // LHS child
            int m1 = n*2 + 2;   // RHS child

            if (m0 < size() && m1 < size())
            {
                if (key[m0] < key[m1])
                {
                    if (key[n] > key[m0])   { key.swap(n, m0); n = m0; }
                    else break;
                }
                else
                {
                    if (key[n] > key[m1])   { key.swap(n, m1); n = m1; }
                    else break;
                }
            }
            else if (m0 < size())
            {
                if (key[n] > key[m0])       { key.swap(n, m0); n = m0; }
                else break;
            }
            else break;
        }
    }

private:
    std::vector<T> key;
};
```



7.2 Btree

Btree can be considered as a variant of binary tree, with 2 main differences (number of children / balance factor) :

- Btree is generalization of binary to M-ary tree (i.e. M children), where M can be odd or even

for M order Btree	*number of keys*	*number of children*	*constraint*
root node	[*1, M-1*]	[2, *M*]	*num of keys = num of children - 1*
general node	[floor((*M*-1)/2), *M*-1]	[floor((*M*+1)/2), *M*]	*num of keys = num of children - 1*
leaf node	[floor((*M*-1)/2), *M*-1]	[*0*]	*num of keys ≠ num of children – 1, num of children = 0*

- Btree is sorted
- keys inside the same node are sorted
- max key in LHS child of this key < this key
- min key in RHS child of this key > this key
- in-order DFS can traverse the key sequentially
- Btree is always balance, i.e. balance factor of each node is zero, implying that all leaf-nodes have ***same depth***
- Insertion in binary tree is a top-down approach, it grows downward from root (hence depth of leaves vary)
- Insertion in Btree is a bottom-up approach, it grows upward from leaves (ensuring all leaves in same depth)
- Insertion in Btree involves promotion of key-value, hence each node should keep a link to parent, i.e. doubly linked.
- Btree and binary tree can achieve $O(\log_M N)$ and $O(\log_2 N)$ search time
- Btree is thinner in tree depth, thus shorter absolute search time, used in database and harddisk
- implement null checking by checking `children`, not by checking `this_node`, unlike binary tree

For example, here is a Btree with $M = 4$, thus for each node, there are [2,4] children and [1,3] `key-value` pairs.

```
                                      k = 100  ← root node
                                      v =   r
                                         |
                 ┌-----------------------┴-------------------------┐
             k = 10/40/65                                       k = 150  ← general nodes
             v =  d/ h/ m                                       v =   v
                  |                                                |
   ┌--------┬-----┴---------┬------------┐              ┌---------┴--------┐
k = 1/5/8  k = 13/24/35  k = 42/47/57/64  k = 67/90/97/99   k = 103/115/135/141  k = 155/170  ← leaf nodes
v = a/b/c  v =  e/ f/ g  v =  i/ j/ k/ l  v =  n/ o/ p/ q   v =   r/  s/  t/  u  v =   w/  x    (having no child)
```

```
template<typename K, typename V> struct node
{
    std::list<std::pair<K,V>> key_values;    // children.size() = key.values.size()+1 [for all non-leaf nodes]
    std::list<node<K,V>*> children;          // children.size() = 0                   [for all     leaf nodes]
    node<K,V>* parent;
};

template<typename K, typename V> struct btree
{
    node<K,V>* root;
};
```

Insertion

To insertion of a new `key-value` pair, for simplicity, suppose the new key does not exist in the tree :

- start off from the root, find a leaf node for adding the new `key-value`
- there exists only one valid leaf for insertion
- general node does not allow insertion of `key-value`
- general node is generated by promotion of `key-value`
- after adding new `key-value` into the leaf node
- if the number of keys in the leaf is less *M-1*, then its done
- if the number of keys in the leaf equals *M*, then perform promotion
- promotion is done by :
- find median `key` and promote it to parent node
- split the `key` vector into two (excluding the promoted median), forming 2 new nodes, each has [floor((*M-1)/2*), *M-1*] keys
- if parent node is `nullptr` (i.e. current node is root), then :
  1. create new parent node
  2. root points to parent node
  3. the parent node will have one key (that is the promoted key)
  4. the parent node will have two children, which are the newly splitted nodes
- promotion is repeated recursively up the tree untill no promotion is needed

Find key

```
template<typename K, typename V>
std::optional<V> btree<K,V>::find(node<K,V>* this_node, const K& key)
{
    if (!children.empty()) // case 1 : for non-leaf node
    {
        auto j = this_node->children.begin();
        for(auto i = this_node->key_values.begin(); i != this_node->key_values.end(); ++i, ++j)
        {
            if      (key <  i->first) return find(*j, key);
            else if (key == i->first) return std::make_optional(i->second);
        }
        return find(*j, key); // the last child
    }
    else // case 2 : for leaf node
    {
        for(auto i = this_node->key_values.begin(); i != this_node->key_values.end(); ++i)
        {
            if      (key <  i->first) return std::nullopt;
            else if (key == i->first) return std::make_optional(i->second);
        }
        return std::nullopt;
    }
}
```

Insertion

Iteration of keys in `insert_to_leaf` is similar to the iteration of keys in `find`. The node pointer to new `key-value` is returned.

```
template<typename K, typename V>
node<K,V>* btree<K,V>::insert_to_leaf(node<K,V>* this_node, const K& key, const V& value) // this_node cannot be nullptr
{
    if (!children.empty()) // case 1 : for non-leaf node
    {
        auto j = this_node->children.begin();
        for(auto i = this_node->key_values.begin(); i != this_node->key_values.end(); ++i, ++j)
        {
            if      (key <  i->first) { return insert_to_leaf(*j, key, value); }
            else if (key == i->first) { i->second = value; return this_node;   }
        }
        return insert_to_leaf(*j, key, value);
    }
    else // case 2 : for leaf node
    {
        for(auto i = this_node->key_values.begin(); i != this_node->key_values.end(); ++i)
        {
            if      (key <  i->first) { this_node->key_value.insert(i, std::make_pair(key, value)); return promote(this_node); }
            else if (key == i->first) { i->second = value; return this_node; }
        }
        this_node->key_value.insert(this_node->key_value.end(), std::make_pair(key, value)); return promote(this_node);
    }
}

template<typename K, typename V>
void btree<K,V>::promote(node<K,V>* this_node)
{
    while(this_node->key_value.size() >= M)
    {
        // lhs_node & rhs_node are created by new operator
        // lhs_node & rhs_node members “key_values” and “children” are filled, while member “parent” is not
        auto [median_key_value_pair, lhs_node, rhs_node] = find_median_and_split(this_node);

        auto parent = this_node->parent; // step 1 : update parent
        if (parent == nullptr)
        {
            parent = new node<K,V>({median_key_value_pair}, {lhs_node, rhs_node}, null_ptr);
            root_node = parent;
        }
        else
        {
            auto i = parent->key_values.begin();
            for(auto j = parent->children.begin(); j != parent->children.end(); ++j, ++i)
            {
                if (*j == this_node)
                {
                    parent->key_values.insert(i, median_key_value_pair);  // if j==children.back(), i==key_values.end()
                    parent->children.insert(j, lhs_node);                 // insert before this_node
                    parent->children.insert(j, rhs_node);                 // insert before this_node
                    parent->children.erase(j);
                    break;                                                // dont forget this
                }
            }
        }
        lhs_node->parent = parent;       // step 2 : update new children
        rhs_node->parent = parent;
        delete this_node;                // step 3 : delete original child
        this_node = parent;
    }
}
```

Function `find_median_and_split` find median key, remove it, split the key-value list into two, split the children list into two. In short, one key is removed, but no child will be removed. Besides, two 2 new nodes are created by `new` operator.

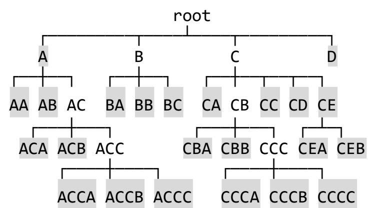
```
template<typename K, typename V>
auto btree<K,V>::find_median_and_split(node<K,V>* this_node)
{
    auto i = this_node->key_values.begin();
    auto j = this_node->children.begin();
    for(int n=0; n!=this_node->key_values.size()/2; ++n) { ++i; ++j; }

    auto lhs_node = new node<K,V>
    {
        { this_node->key_values.begin(), i },
        { this_node->children.begin(), j+1 },
        nullptr // parent is set in step 2 of promote()
    };
    auto rhs_node = new node<K,V>
    {
        { i+1, this_node->key_values.end() },
        { j+1, this_node->children.end()   },
        nullptr // parent is set in step 2 of promote()
    };
    return std::make_tuple(*median_key_value_iter, lhs_node, rhs_node);
}
```

Traversal
It's a generalization of traveral in binary tree.

```
template<typename K, typename V>
void btree<K,V>::DFS_recursion(node<K,V>* this_node, std::function<void(const K&, const V&)>& fct, int mode)
{
    if (children.empty())
    {
        for(auto& x:key_values) fct(x.first, x.second);
    }
    else if (mode == pre_order)
    {
        for(auto i = key_values.begin(); i != key_values.end(); ++i) fct(i->first, i->second);
        for(auto j =   children.begin(); j !=   children.end(); ++j) DFS_recursion(*j, fct, mode);
    }
    else if (mode == in_order)
    {
        auto j = children.begin();
        DFS_recursion(*j, fct);
        for(auto i = key_values.begin(), ++j; i != key_values.end(); ++i, ++j)
        {
            fct(i->first, i->second);
            DFS_recursion(*j, fct);
        }
    }
    else
    {
        for(auto j =   children.begin(); j !=   children.end(); ++j) DFS_recursion(*j, fct, mode);
        for(auto i = key_values.begin(); i != key_values.end(); ++i) fct(i->first, i->second);
    }
}
```

7.3 Prefix tree (also known as Trie)

- key is not necessary a string, can be any sequence (but for simplicity, we assume string for now)
- key of this node is the key of its parent concaternating with one extra character
- key of this node is stored in its parent (not in this node)
- depth of this node is the length its key, prefix tree is not balanced
- all leaves must have a value, but intermediate nodes like `CB` or `CCC` may *NOT* have a value
- implemented by merging `btree<K,V>::key_values` and `btree<K,V>::children` into one map
- implement null checking by checking `children`, not by checking `this_node`, unlike binary tree



node `CB` has a value iff `insert("CB",value)` has been called

bolded nodes below are leaves

```
template<typename V> struct node
{
    std::optional<V> value; // note : some nodes may have no value
    std::map<char, node<V>*> children;
};

template< typename V> struct prefix_tree
{
    node<V>* root;
};
```

Lets implement `find`, `insert` and `DFS_recursion_inorder`. There are 4 cases : `key` is empty or not, `children` is empty or not.

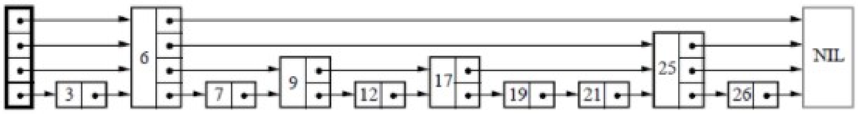
```
template<typename V> std::optional<V> prefix_tree<V>::find(node<V>* this_node, const std::string& key)
{
    if (key.empty())
    {
        return this_node->value;
    }
    else if (auto i = this_node->children.find(key[0]),
              i!= this_node->children.end())
    {
        return find(i->second, key.substr(1));
    }
    else
    {
        return std::nullopt;
    }
}

template<typename V> node<V>* prefix_tree<V>::insert(node<V>* this_node, const std::string& key, const V& value)
{
    if (key.empty())
    {
        this_node->value = std::make_optional(value);
        return this_node;
    }
    else if (auto i = this_node->children.find(key[0]),
              i!= this_node->children.end())
    {
        return insert(i->second, key.substr(1), value);
    }
    else
    {
        auto new_node = new node<V> {{},{}};
        this_node->children[key[0]] = new_node;
        return insert(new_node, key.substr(1), value);
    }
}

// Pre-order traversal gives dictionary-like ordering.
template<typename V> void prefix_tree<V>::DFS_recursion_preorder(node<V>* this_node, const std::string& this_node_key,
                                                                 std::function<void(const std::string&, const V&)>& fct)
{
    if (this_node->value) fct(key, *(this_node->value));
    for(const auto& x:this_node->children)
    {
        std::string child_node_key = this_node_key + std::string(x.first, 1); // construct string from 1 char
        DFS_recursion_preorder(x.second, child_node_key, fct);
    }
}

// This is how DFS is invoked :
DFS_recursion_preorder(root, "", fct); // key of root node is empty
```

7.4 Skip list (this part is not reviewed yet)

Please read the original paper Skip Lists : A Probabilistic Alternative to Balanced Trees, William Pugh

Skip list is another variant of binary tree.

- Like both binary tree and Btree : all nodes are sorted
- Like Btree :
- all leaves have same depth
- all nodes are grown in bottom-up approach *(in a probabilistic manner)*
- Unlike both binary tree and Btree :
- all `(key,value)` pairs exist in the bottom layer (never in the middle layers) i.e. all are leaf-nodes
- all non-leaf nodes are just short cut to leaf-nodes to squeeze search time to *Olog(N)*
- there is link between parent-and-child (vertical direction)
- there are links across sibling-nodes in the same layer (horizontal direction) *[there is no link between siblings in binary tree]*



Randomized tree

After insertion and deletion of in binary tree or Btree, rebalancing is needed. Skip list avoids complicated rebalancing by adopting a randomized approach. Given a skip list parameter *p* *(0.5 by default)* which is the probability of have a shortcut in parent layer, and if bottom layer is labelled as layer *0*, its parent is layer *1*, and so on (suppose the top layer is layer *M-1*) we have :

```
// In a long run, skip-list is closed to balanced (on average).
std::uint32_t randomize_number_of_layer_of_shortcut()
{
    std::uint32_t n = 0;
    while(true && n < MAX_POSSIBLE_LAYERS)
    {
        auto r = rand()%10000 / 10000.0;
        if (r > p) ++n;
        else return n;
    }
    return n;
}
```

```
expected[num_nodes_in_layer(M-1)] = 1 (just expectation, not realize)
expected[num_nodes_in_layer(M-2)] = 2
expected[num_nodes_in_layer(M-3)] = 4
...
expected[num_nodes_in_layer(2)]   = M-2
expected[num_nodes_in_layer(1)]   = M-1
expected[num_nodes_in_layer(0)]   = M (the real underlying list)
```

Node definition

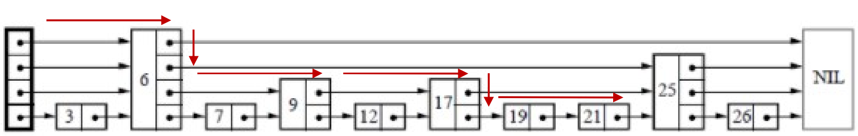
Here is the definition of a node. Practically, it is implemented as a list rather than a tree, by putting all pointers from non-leaf layers to the same leaf together, in the same `struct`. Thus effectively, there is no traversal from parent to child, just traversal across siblings in same layer. Different nodes have different `shortcuts.size()`.

```
template<typename K, typename V> struct node
{
    K key;
    V value;
    std::vector<node<K,V>*> shortcuts; // Note : These are not links to children, they are shortcuts to siblings.
};
```

Search

Searching is a near-binary search, which starts from the root. For example, if we want to find `key=21` in the following, the red path is the resulting traversal in the skip list. Its like 2D traversal on a sparse matrix in diagonal direction.

- if `key` is identical to next node in same layer, return the `value`
- if `key` is greater than next node in same layer, take a horizontal traverse
- if `key` is smaller than next node in same layer, take a vertical traverse

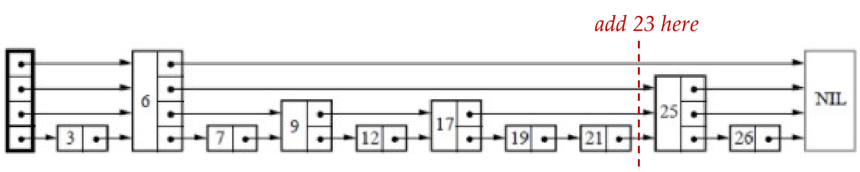


For finding a target key, the blue code lines are good enough, however, if we want to reuse this `find` function inside `insert`, we need to record the turning points in the red path, this is the purpose of the red code lines.

```
// Starting from the root ...
template<typename K, typename V> node<K,V>* skip_list<K,V>::find(const K& key, std::vector<node<K,V>*>& path);
{
    auto this_node = root;
    std::uint32_t layer = root->shortcuts.size()-1;
    path.clear();
    path.resize(root->shortcuts.size());

    while(layer > 0)
    {
        if      (this_node->shortcuts[layer] == nullptr) { path[layer] = this_node;         --layer;  }
        else if (this_node->shortcuts[layer]->key > key) { path[layer] = this_node;         --layer;  }
        else if (this_node->shortcuts[layer]->key < key) { this_node   = this_node->shortcuts[layer]; }
        else                                             { path[layer] = this_node; return this_node; }
    }
    while(this_node != nullptr)
    {
        if      (this_node->key > key)                   { return std::nullptr;                       }
        else if (this_node->key < key)                   { this_node   = this_node->shortcuts[0];     }
        else                                             { path[layer] = this_node; return this_node; }
    }
    // Redundancy : return value == path[0] is always true.
    return std::nullptr;
}
```

Insertion

Insertion involves calling `find_before` function, which is a modified version of `find`. The function `find_before` finds the last node just smaller than the target `key`. Suppose we are going to add 23 :




```
template<typename K, typename V> node<K,V>* skip_list<K,V>::insert(const K& key, const V& value)
{
    // Step 1 : get the path
    std::vector<node<K,V>*> path;
    find_before(key, path);

    // Step 2 : get the random number
    auto num_layer = randomize_number_of_layer_of_shortcut();

    // Step 3 : create new node
    node<K,V>* this_node = new node<K,V>(key, value);
    this_node->shortcuts.resize(num_layer);

    // Step 4 : update links across the above red dotted line
    for(std::uint32_t n=0; n!=num_layer; ++n)
    {
        this_node->shortcuts[n] = path[n]->shortcuts[n]; // may be std::nullptr, its ok ...
        path[n]->shortcuts[n] = this_node;
    }
}
```

Deletion

Deletion is similar to insertion without *step 2*, replace *step 3* by `delete operator`.

8. Graph and DAG

Directional acyclic graph sorting

Directional acyclic graph (DAG) is a directed graph without cycles. Topological sorting is defined as a sorting, in which vertex V_n is always ordered before V_m if there exists a directed edge E_{nm}.

- Define S_1 as set of dependent nodes, which is initialized as all nodes in the graph.
- Define S_2 as set of independent nodes, which is initially empty (nodes without ***unvisited*** incoming edges are independent).
- Define S_3 as ordered set of nodes in topological order, i.e. the output.

$$V = S_1 \cap S_2 \cap S_3$$
$$\varnothing = S_1 \cup S_2 \cup S_3$$

It is done by region growing :

- pop a node from S_2 and push it to S_3
- mark the edges to all its neighbours in S_1 as ***visited***
- move neighbours becoming independent from S_1 toS_2
- nodes are effectively moving from S_1 to S_2 and finally to S_3
- repeat the above until S_1 and S_2 are empty

Dijkstra shortest path algorithm

Given a graph, with a subset of source nodes and a subset of destination nodes, find the shortest path from source to destination, as distance between nodes are specified in each edge. The solution is also a region-growing algorithm.

- For each node in the graph, define 2 items : `dist` and `prev`,
- the former measures the minimum distance from any source to this node
- the latter stores the previous node for backtracking
- We use priority queue for the growing process.
- push all nodes into priority queue, `dist` is set zero for source, and infinite for other nodes
- pop node *n* from priority queue, update its neighbour *m* if `dist[m]` can be shortened
- that is if `dist[n]+edge[n,m] < dist[m]`, then update `dist[m] = dist[m]+edge[n,m]` and `prev[m] = n`
- repeat the above step until one destination is popped
- The priority queue is a special one (not `std::priority_queue`), because we need to :
- access a node, change the node value, and trigger ***floating*** or ***sinking***

Disjoint sets (Union find algorithm)

Given an undirected graph with vertex set $V = \{A, B, C, ...\}$ and edge set E, two nodes are said to be in the same subset if there exists at least one path connecting these two nodes. The graph can be represented as disjoint set, each set is labelled or represented by one of the node in a set. The problem is to implement two basic functions :

1. `find(v)` which returns the representative of vertex `v`, `find(vertex)` can be used to implement `is_same_set(v0,v1)`
2. `union(v0,v1)` which joins the set where `v0` belongs with the set where `v1` belongs together, so that `is_same_set(u0,u1) = true`
   for all `is_same_set(u0,v0) = true` and
   for all `is_same_set(u1,v1) = true`

Example of disjoint set is "unit conversion" problem. Here is the naïve implementation.

- each set is represented as a tree, rather than a graph (or complete graph), for simple implementation
- each set is represented (labelled) by the root node of the tree
- we need to traverse the tree upward only for `union` and `find`, so tree is implemented with `parent` link but NO `children` link
- `parent` link can be simply implemented as a `std::unordered_map<V,V>` where key is vertex, value is its parent
- root is denoted as the vertex having itself as its parent `parent[v]=v`
- at the beginning all nodes are disjoint, then apply `union` for each edge to construct the `disjoint_sets`

Implementation of disjoint sets

```
template<typename V> class disjoint_sets
{
public:
    disjoint_sets(const std::set<V>& vertices)
    {
        for(const auto& v:vertices) parent[v] = v;
    }

    // find() means finding representative
    const V& find_recursive(const V& v) const noexcept
    {
        if (parent[v] == v) return v;
        return find(parent[v]);
    }

    const V& find_iterative(const V& v) const noexcept
    {
        auto v0 = v;
        while(parent[v0] != v0) v0 = parent[v0];
        return v0;
    }

    bool is_same_set(const V& v0, const V& v1)
    {
        auto r0 = find(v0);
        auto r1 = find(v1);
        return r0 == r1;
    }

    void union(const V& v0, const V& v1) noexcept
    {
        auto r0 = find(v0);
        auto r1 = find(v1);
        if (r0 != r1) parent[r1] = r0; // remark : NOT parent[v1] = v0 (BUG!!!)
    }

private:
    std::unordered_map<V,V> parent;
};
```

If `union` is implemented by `parent[v1] = v0`, then there will be bug. Considered the following case, arrow denotes link to parent.

```
A <- B <- C
D <- E <- F
```
then adding `edge(C,F)` will result in problem in `parent` map

The above implementation of `find` and `union` is *O(logN)* on average, where *N* is number of vertex. However if trees are not balanced, the computation time will become *O(N)*. In order to speed up, we can apply method called path compression to `find`, which makes trees as flat as possible. Path comparison by calling `find(v)` means converting every node in the path "from root to the vertex `v`" to a direct child of the root. Path compression can be done by :

```
    const V& find_recursive_with_path_compression(const V& v) const noexcept
    {
        if (parent[v] == v) return v;
        return parent[v] = find(parent[v]); // please check if C++ supports this syntax
    }
```

Comparing the 2 implementations, the second one update the `parent` while returning root of tree. Therefore, as we call the function more often, the tree will become flatter and more efficient. Besides as the implementation is recursive, which makes use of call stack, path compression starts from the top of tree to the leaf of tree. For example, when we apply `find(I)`

```
    A                       A                     A                     A
    |                      / \                   /|\                   /|\ \
    B                     B   D                 B D F                 B D F I
   / \                   /   / \               /  | |\               /  |   \
  C   D                 C   F   G             C   G H I             C   G   H
 /   / \               /   / \   \           /    |                /    |
E   F   G             E   H   I   J         E     J               E     J
   / \   \
  H   I   J
```

By expanding the function sequence in recursion, we have :

`find(A)` returning `A`
`find(B)` returning `A` and assigning `parent[B] = A` (no change in tree)
`find(D)` returning `A` and assigning `parent[D] = A`
`find(F)` returning `A` and assigning `parent[F] = A`
`find(I)` returning `A` and assigning `parent[I] = A` (the tree will be completely flat if we call `find()` on `E`, `H` and `J`)

9. Sorting Algorithms – Ascending order sorting

Sorting algorithm can be characterized by its speed, inplace (no auxiliary container needed) and stability. A stable sorting means a sorting which maintains the relative order of items having the same key, before and after sorting. Sorting is useful, because a search in an unsorted array needs *O(N)* in time, while a search in a sorted array needs *O(logN)* only, as bisection kicks in.

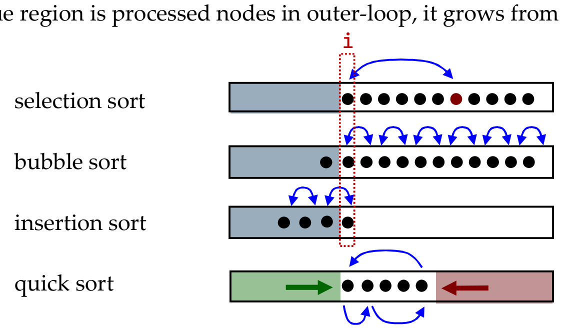
Selection, bubble and insertion sort

- 3 sortings involve double for-loop :
- outer for-loop iterates in forward direction, using index `i`
- inner for-loop iterates in forward direction for selection-sort, backward direction for insertion and bubble sort, using index `j`
- inner for-loop for the 3 sortings :
- selection sort inner for-loop forward scans `j∈[i,N-1]`, picks the minimum, swap it with `element[i]`
- bubble sort inner for-loop backward scans `j∈[i+1,N-1]`, swap the smaller one in `element[j-1]` or `element[j]` forward
- insertion sort inner for-loop backward scans `j∈[0,i-1]`, insert `element[j]` in right place and skip the rest

Quicksort and mergesort (both are divide and conquer, lets talk about the former)

- In each recursion, quicksort divides the container into three parts : `partA`, `unclassified` and `partB`
- the objective to put `element[i]` in the right place so that there is no need to move it again
- compare the first node (reference) with the last node in unclassified-part :
- if they are in-order, then move the last node to `partB` (in fact all we need to do is to move `partB.begin`)
- if they are not, then move the last node to `partA` (which involves a 3-element swap)
- effectively, by taking first node in `unclassified` as reference, we classify nodes in `unclassified` into `partA` and `partB`
- repeat until `unclassified` is gone and trigger ***recursion*** on `partA` and `partB`
- performs better when `partA/B` are equal sized, but $O(N^2)$ when input is sorted or inversely sorted

Blue region is processed nodes in outer-loop, it grows from *LHS* to *RHS*. Black nodes are under-processing in current inner-loop.



Sorting that involves `std::swap(x,y)` where no comparation `comp(x,y)` has been done will make the sorting algorithm unstable.

like 3way-swap in quicksort, float/sink in heap

	best	average	worst	inplace	stable	iterator	concept
selection sort	n^2	n^2	n^2	yes	no	forward	
bubble sort	n	n^2	n^2	yes	yes	bidirectional	
insertion sort	n	n^2	n^2	yes	yes	bidirectional	
quick sort	$nlogn$	$nlogn$	n^2	yes	no	bidirectional	divide conquer
merge sort	$nlogn$	$nlogn$	$nlogn$	no	yes	forward	divide conquer
heap sort	$nlogn$	$nlogn$	$nlogn$	yes	no	random access	binary heap
pigeon hole sort	-	$n + 2^k$	-	-	-	forward	histogram
topological sort	-	$V + E$	-	-	-	-	region growing

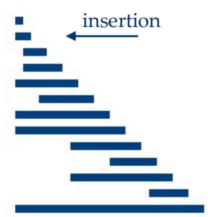
User can provide comparator as a functor.

```
template<typename T> struct less : public std::binary_function<T,T,bool>
{
    bool operator()(const T& x, const T& y) const { return x < y; }
};
```

The following implementation does not consider empty case.


```
template<typename ITER>
void select_sort(ITER begin, ITER end)
{
    for(ITER i=begin; i!=end; ++i)
    {
        ITER min = i;
        for(ITER j=i+1; j!=end; ++j) if (*j < *min) min = j;
        std::swap(*i,*min);
    }
}
```


```
template<typename ITER>
void bubble_sort(ITER begin, ITER end)
{
    for(ITER i=begin; i!=end; ++i)
    {
        for(ITER j=end-1; j!=i; --j) if (*j < *(j-1)) std::swap(*j,*(j-1));
    }
}
```




```
template<typename ITER>
void insert_sort(ITER begin, ITER end)
{
    for(ITER i=begin; i!=end; ++i)
    {
        for(ITER j=i; j!=0; --j) { if (*j < *(j-1)) std::swap(*j,*(j-1)); else break; }
    }
}

template<typename ITER>
void quick_sort(ITER begin, ITER end)
{
    ITER i = begin;
    ITER j = end-1;
    while(i!=j) // It can handle case with i+1==j, the following swap becomes 2-element-swap.
    {
        if (*i < *j) --j;
        else
        {
            auto temp = *i; // We can either start with temp = *i or temp = *j. NOT temp = *(i+1), last one cant handle i+1=j
            *i = *j;
            *j = *(i+1);
            *(i+1) = temp;
            ++i;
        }
    }
    quick_sort(begin, i);
    quick_sort(i+1, end);
}
```

For merge sort, deep copy of half of the elements is inevitable.

```
template<typename ITER> void merge_sort(ITER begin, ITER end)
{
    typedef typename std::iterator_traits<ITER>::value_type T;
    typedef typename std::back_insert_iterator<std::vector<T>> BACK_INS;
    typedef std::vector<T> aux_vec;

    ITER mid = begin;
    std::advance(mid, (std::distance(begin, end)+1)/2);            // +1 is used to ensure 1st half longer than 2nd half
    if (mid == begin) return;                                      // i.e. return when size is 0 or 1
    std::copy(begin, mid, std::back_inserter(aux_vec));            // back_inserter is a function that returns BACK_INS

    merge_sort(mid, end);
    merge_sort(aux_vec.begin(), aux_vec.end());
    merge(aux_vec.begin(), aux_vec.end(), middle, end, begin);     // ensure 1st half is longer than 2nd half ...
}                                                                  // otherwise output vector is overwritten

// Merge sort depends on the following merge algorithm.
template<typename ITER, typename BACK_INS>
void merge(ITER begin0, ITER end0, ITER begin1, ITER end1, BACK_INS out)
{
    ITER x = begin0;
    ITER y = begin1;
    while(x!=end0 && y!=end1)
    {
        if (*x < *y)  { *out = *x; ++x; ++out; }
        else          { *out = *y; ++y; ++out; }
    }
    while(x!=end0)    { *out = *x; ++x; ++out; }
    while(y!=end1)    { *out = *y; ++y; ++out; }
}
```

For heap sort implemented inplace, we need a **factory** that can construct inplace heap from iterator pairs, which is possible, but we need to rewrite our heap in previous section.

```
template<typename ITER, typename COMP = std::less<typename std::iterator_traits<ITER>::value_type>>
void heap_sort(ITER begin, ITER end)
{
    inplace_heap<typename std::iterator_traits<iter_type>::value_type, COMP> heap = inplace_heap_factory(begin, end);

    for(ITER i=begin; i!=end; ++i)
    {
        *i = heap.top();
        heap_object.pop();
    }
}
```

There are still bugs in the above implementation, the first item is poped from `heap` and overwrite the `begin`, so the `heap` is overwriting itself. There are 2 solutions.

Solution 1
Sort by `std::less`, with the inplace heap's `top` started `end-1` to `begin`. When the first item is popped, the inplace heap occupies from `end-1` to `begin+1`, we can then overwrite `begin` with the first popped (least) value. And repeat the step. We need to implement `inplace _heap_factory` in a specific way.

Solution 2
Sort by `std::greater`, with the inplace heap's `top` started `begin` to `end-1`. When the first item is popped, the inplace heap occupies from `begin` to `end-2`, we can then overwrite `end-1` with the first popped (greatest) value. And repeat the step. We need to infer `std::greater` from `std::less` with template.