Algorithms and Data Structures

Author: Daria Shutina

Organization stuff

Professor Kinga Lipskoch

- Office: Research I, Room 94
- Telephone: +49 421 200-3148
- E-Mail: klipskoch@constructor.university
- Office hours: Mondays 10:00 - 11:00

Lecture slides: https://grader.eecs.jacobs-university.de/courses/ch_231_a/2023_1/

Homeworks:

Use Grader for homework submission. Do not forget to change semester from Fall 2022 to Spring 2023.

Submit ZIP file containing one PDF file and source code fileswith makefile.

Homeworks are not mandatory.

VPN for remote access to Jacobs Net:

Source: https://teamwork.jacobs-university.de/display/ircit/VPN+Access

Domain name: vpnasa.jacobs-university.de (Jacobs VPN)

Tutorials:

- 2 weekly tutorials given by one TA
- Tutorial before homework deadline
- Online via Teams, Saturdays, 19:00 – 21:00
- Online via Teams, Sundays, 19:00 – 21:00

23-02-02

Templates

Templates allow to write generic code, i.e., code which will work with different types. Later a specific type will be provided and the compiler will substitute it.

Motivation for using templates is to eliminate snippets of code that differ only in the type and do not influence on the logic.

While using templates, operators for some types need to be overloaded.

Why function definitions are written in `.h` files?

Both declaration an definition are written in `.h` files! Otherwise, the code will fail. The reason is that, when instantiating a template, the compiler creates a new class with a definite type. Consequently, it has to instantiate methods. But it cannot find methods with an appropriate type, since they are written in another `.cpp` file.

If you still want to have declaration and definition of a template in different files, there are some options (but think twice).

One possible solution is to instantiate a template in the same file where methods are written:

```cpp
// a.cpp
template <class T>
void MyTemplate<T>::do() {
    // do sth
}

template class MyTemplate<int>;
template class MyTemplate<std::string>;
```

Another solution is to include a `.cpp` file inside the header file (wtf??). Btw, you do not have to compile this file with others:

```cpp
// a.h
template <class T>
class MyTemplate {
    T prop;
public:
    void do() {}
}

#include a.cpp
```

Basic syntax

Type parameterization in classes:

```
template <class T>
class MyClass {
    T i;
public:
    MyClass() = default;
    MyClass(T arg) : i(arg) {}
};

int main() {
    MyClass<int> aboba;
}
```

Type parameterization in functions:

```
template <class T>
void my_function(T arr[], int size) {
    for (int i = 0; i < size; ++i) {
        std::cout << arr[i] << ' ';
    }
}
```

Note: `class` in `<class T>` can be replaced with `typename`.

Overloading an operator

```
template <class T>
class MyClass {
    T array[size];
public:
    BoundedArray(){};
    T& operator[](int);
};

...

```

```cpp
template<class T>
T& MyClass<T>::operator[](int pos) {
    if ((pos < 0) || (pos >= size))
        exit(1);
    return array[pos];
}
```

23-02-07

Fundamental Container Classes

- Sequence containers
    - vector
        - fast random access
        - fast inserting/removing at the end, slow inserting/removing in the middle
        - not efficient while resizing
    - deque
        - a dynamic array which can grow in both directions
        - random access with a constant complexity
        - fast inserting/removing at beginning/end, slow inserting/removing at the middle
    - list
        - no random access
        - fast inserting/removing at beginning/end, slow inserting/removing at the middle
        - In a linked list, there can be *sentinels* -- markers that are used to represent the beginning or end of a data structure.
- Associative containers
    - set
        - implemented as a BST
        - sorted
        - duplicates are not allowed
    - multiset
        - as a set, but duplicates are allowed

        - map
            - stores `key/value` pairs
            - the key is the basis for ordering
            - keys are not duplicated
            - called "associative array"
        - multimap
            - as a map, but keys can be duplicated
            - called "dictionary"
- Container adapters
    - stack (can be implemented as an array)
    - queue (can be thought of as a circle with n segments, somewhere is head, somewhere is tail. can be implemented as an array)
    - priority queue

Set operations

For a set, you can use such operations as

- `set_union(...)` -- $A \cup B$
- `set_difference(...)` -- $A \backslash B$
- `set_intersection(...)` -- $A \cap B$
- `set_symmetric_difference(...)` -- $(A \backslash B) \cup (B \backslash A)$

23-02-14

decltype vs auto

Both tools are used in order to avoid writing long types.

The `decltype` keyword allows to obtain the type of an expression at compile-time. For example, it is useful with templates, when the type of the return result depends on the type of the template.

```
#include <bits/stdc++.h>

int fun1() { return 10; }
char fun2() { return 'g'; }

```

```
int main() {
    int x = 42;
    decltype(x) y = 10;  // y has the same type as x (int)

    // type of `x` is `int`, since return type of `fun1` is int
    decltype(fun1()) x;

    // type of `y` is `char`, since return type of `fun2` is char
    decltype(fun2()) y;

    std::cout << typeid(x).name() << std::endl;  // i
    std::cout << typeid(y).name() << std::endl;  // c
}
```

```
template <typename T>
void printType(const T& value) {
    decltype(value) x = value;  // x has the same type as value
    std::cout << typeid(x).name() << std::endl;
}
```

The `auto` keyword specifies that the type of the variable that is being declared will be automatically deducted from its initializer. The variable declared with auto keyword should be initialized at the time of its declaration, otherwise there will be a compile error.

```
#include <bits/stdc++.h>

int main() {
    set<string> st = { "ab", "ob", "a" };
    auto it = st.begin();
    std::cout << *it << std::endl;  // a
}
```

Attention! `auto` emits reference (and `decltype` does not). If you want something like `int&`, you need to write `auto&`.

Lambda functions

```
auto my_lambda = [ /*captureList*/ ] ( /*params*/ ) { ... }
```

Options for the capture list:

- capture by value

```cpp
int num = 100;

auto my_lambda = [num] (int val) {
  return num + val;
};

std::cout << my_lambda(100) << '\n'; // 200
std::cout << num << '\n';            // 100
```

- capture by reference

```cpp
int num = 100;

auto my_lambda = [&num] (int val) {
  num += val;
  return num;
};

std::cout << my_lambda(100) << '\n'; // 200
std::cout << num << '\n';            // 200
```

Variadic templates

Variadic templates are templates that can take zero or positive number of arguments.

```cpp
#include <iostream>

void print() {
    std::cout << "I am empty function\n";
}

// Variadic function Template that takes variable
// number of arguments and prints all of them.
template <typename T, typename... Types>
void print(T var1, Types... var2) {
    std::cout << var1 << std::endl;
    print(var2...);
}

int main() {
    print(1,
          2,
          3.14,
          "Pass me any number of arguments",
          "I will print\n");
```

```
}
```

Variadic functions

In comparison with variadic templates, variadic functions have a fixed amount and a fixed type of given arguments.

You can use `cstdarg` library for variadic functions. There are library functions such as `va_list`, `va_start`, `va_copy`, `va_end`, etc.

```cpp
#include <iostream>
#include <cstdarg>

// `count` is the number of parameters
// `...` reflect the fact that the function is variadic
double average(int count, ...) {
    va_list ap; // the list of given arguments
    int j;
    double sum = 0;

    va_start(ap, count);
    for (j = 0; j < count; j++) {
        sum += va_arg(ap, double); // gets one argument and
                                   // increments `ap` to the next argument.
    }
    va_end(ap);

    return sum / count;
}

int main() {
    std::cout << average(4, 1.0, 2.0, 3.0, 4.0) << '\n';
    std::cout << average(2, 3.0, 5.0) << '\n';
    std::cout << average(5, 3.2, 3.2, 3.2, 3.2, 3.2) << '\n';
    return 0;
}
```

23-02-16

Analysis of time complexity

Types of analisys:

- worst case
- average case

Cheat sheet with time complexities: https://www.bigocheatsheet.com/

Asymptotic Analysis

$$f = \Theta(g) \Leftrightarrow \exists n_0, c_1 > 0, c_2 > 0 \ : \ c_1 \cdot g(n) \leqslant f(n) \leqslant c_2 \cdot g(n), \forall n > n_0.$$

Upper bound:

$$f = O(g) \Leftrightarrow \exists n_0, c > 0 \ : \ 0 \leqslant f(n) \leqslant c \cdot g(n), \forall n > n_0.$$

Lower bound:

$$f = \Omega(g) \Leftrightarrow \exists n_0, c > 0 \ : \ 0 \leqslant c \cdot g(n) \leqslant f(n), \forall n > n_0.$$

Non-tight upper bound:

$$f = o(g) \Leftrightarrow \forall c > o \, \exists n_o \ : \ 0 \leqslant f(n) < c \cdot g(n), \forall n > n_0$$
$$\Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Non-tight lower bound:

$$f = \omega(g) \Leftrightarrow g = o(f)$$
$$\Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

Computation with limits

$f \in o(g)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$f \in O(g)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
$f \in \Omega(g)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$
$f \in \Theta(g)$	$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
$f \in \omega(g)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

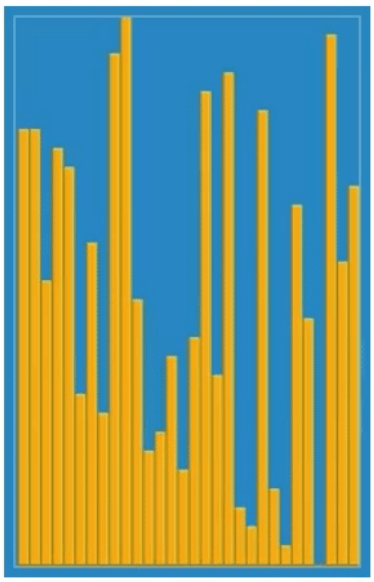
23-02-21

Sorting algorithms

A *stable sort* is a sorting algorithm which does not change the order of values if their keys are the same. For example, Insertion sort is stable. Counting sort is stable if the order is descending in the last iteration.

Insertion sort

Go from 0 to $n-1$. Consider a current element $a[k]$ and insert it in $a[0..k-1]$.



```
INSERTION-SORT(A, n)
 for j = 2 to n
     key = A[j]
     // Insert A[j] into the sorted sequence A[1 .. j − 1].
     i = j − 1
     while i > 0 and A[i] > key
         A[i + 1] = A[i]
         i = i − 1
     A[i + 1] = key
```

Time complexity:

$O(n^2)$. We make $n-1$ iterations in total. Imagine that, on every iteration, we have to compare an element with all elements in front of it. Thus, we have $1+2+\ldots+(n-1)=\frac{n(n-1)}{2}=O(n^2)$ comparisons in total.

Correctness:

base $n=1$. An array of size 1 is sorted.

Transition: $a[0..k-1]$ is sorted. We consider an element $a[k]$ and insert it between $a[i]$ and $a[i+1]$ so that $a[i] \leqslant a[k] \leqslant a[i+1]$. Thus, we get a sorted array $a[0..k]$.

Merge Sort

Recursively sort $a[0..\frac{n}{2})$ and $a[\frac{n}{2}..n)$. Then merge them.

```cpp
void merge(std::vector<int> &a, int left, int mid, int right)
{
    int leftSubArraySize = mid - left;
    std::vector<int> leftSubArray(leftSubArraySize);
    for (int i = 0; i < leftSubArraySize; ++i) {
        leftSubArray[i] = a[i + left];
    }

    int rightSubArraySize = right - mid;
    std::vector<int> rightSubArray(rightSubArraySize);
```

```
    for (int i = 0; i < rightSubArraySize; ++i) {
        rightSubArray[i] = a[i + mid];
    }

    int indexOfRightSubArray = 0;
    int indexOfGivenArray = left;

    for (int indexOfLeftSubArray = 0; indexOfLeftSubArray < leftSubArraySize;
indexOfLeftSubArray++) {
        while (indexOfRightSubArray < rightSubArraySize
               && rightSubArray[indexOfRightSubArray] <=
leftSubArray[indexOfLeftSubArray]) {
            a[indexOfGivenArray++] = rightSubArray[indexOfRightSubArray++];
        }
        a[indexOfGivenArray++] = leftSubArray[indexOfLeftSubArray];
    }

    while (indexOfRightSubArray < rightSubArraySize) {
        a[indexOfGivenArray++] = rightSubArray[indexOfRightSubArray++];
    }
}

void mergeSort(std::vector<int> &a, int left, int right) {
    if (right - left <= 1) return;
    int mid = left + (right - left) / 2;
    mergeSort(a, left, mid);
    mergeSort(a, mid, right);
    merge(a, left, mid, right);
}
```

Time complexity:

$$T(n) = \begin{cases} O(1),\ if\ n = 1 \\ T(\frac{n}{2}) + \Theta(n),\ if\ n > 1 \end{cases}$$

$T(n)$ is divided by 2, so we can build a recursion tree, which height is $O(\log n)$ and the amount of leaves is $O(n)$ $\Rightarrow$ total time complexity is $O(n \log n)$.

Heap sort

Используем бинарную кучу с наибольшим элементом в корне. `a[1]` -- корень.

Заполняем массив в правильной последовательности с конца. Меняем первый элемент с последним, потом проталкиваем "новый" первый элемент на нужное место.

Time complexity is $O(n \log n)$.

```cpp
void heapSort(vector<int> &a) {
    int n = a.size();
    build(a, n);
    for (int i = n; i >= 2; --i) {
        std::swap(a[1], a[i]);
        n--;
        maxHeapify(a, n, 1);
    }
}
```

Quicksort

Using Divide & Conquer idea.

Divide an array into two subarrays around a pivot x such that the elements in left subarray are $\leqslant x$, and elements in right subarray are bigger. Then recursively sort subarrays.

```cpp
int partition(vector<int> &a, int left, int right) {  // O(n)
    int x = a[left];
    int i = left;

    for (int j = left + 1; j <= right; ++j)
        if (a[j] <= x) {
            i++;
            swap(a[i], a[j]);
        }

    swap(a[left], a[i]); // чтобы `x` разделял массив,
                         // а не был в начале
    return i;
}

void quickSort(vector<int> &a, int left, int right) {
    if (left < right) {
        int i = partition(a, left, right);
        quickSort(a, left, i);
        quickSort(a, i + 1, right);
    }
}

quickSort(a, 0, n - 1);
```

In the worst case, time complexity is $O(n^2)$. On average, it is $O(n \log n)$.

If a pivot is chosen randomly, it helps prevent worst-case scenarios where the array is already sorted or nearly sorted.

Counting sort

Given an array `A`, find a sorted array `B`.

Array `C` is kinda storage.

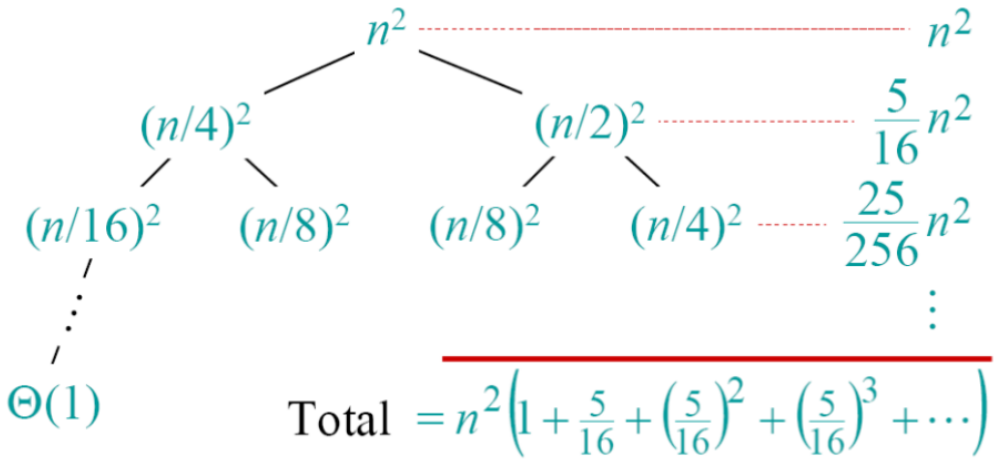
```cpp
void countingSort(vector<int> &a, vector<int> &b, int n) {
    int k = max(a); // max element in array `a`
    vector<int> c(k);

    for (int i = 0; i < n; ++i)
        c[a[i]]++;  // c[s] = # of elems t \in a : t = s

    for (int i = 2; i < k; ++i)
        c[i] += c[i - 1];  // c[s] = # of elems t in a : t <= s

    for (int i = 0; i < n; ++i) {
        b[c[a[i]] - 1] = a[i];
        c[a[i]]--;
    }
}
```

Counting sort can be inefficient, if k (the max value in `A`) is large.

Recurrence Solving Methods

Recursion tree

$$T(n) = \begin{cases} T(\frac{n}{4}) + T(\frac{n}{2}) + n^2, \ n > 1 \\ \Theta(1), \ if \ n = 1 \end{cases}$$



$n^2(1 + \frac{5}{16} + (\frac{5}{16})^2 + \ldots) = O(n^2)$.

Substitution Method

1. Guess the form of the solution
2. Verify by induction (assume that $T(n)$ can be limited by $c \cdot n^k$ with $c, k \in R$)
3. Solve for constants

Example

$T(n) = 4T(\frac{n}{2}) + n$ with base case $T(1) = \Theta(1)$.

Our guess is $O(n^3)$.

$T(n) \leqslant 4c \cdot (\frac{n}{2})^3 + n = \frac{c}{2} \cdot n^3 + n \leqslant cn^3$ with $c \geqslant 2$

$O(n^2)$ is not enough. Assume, $T(n) \leqslant 4c \cdot (\frac{n}{2})^2 + n = cn^2 + n$. It is always greater than cn^2.

Master method

Applied to recurrences of the form $T(n) = aT(\frac{n}{b}) + f(n)$

Complexity depends on comparing $f(n)$ with $n^{\log_b a}$.

1. If $\exists c > 0 \ : \ f(n) = O(n^{\log_b a - c}) \implies T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a}) \implies T(n) = \Theta(n^{\log_b a} \log n)$
3. If $\exists c > 0 \ : \ f(n) = \Omega(n^{\log_b a + c})$ and $\exists c' < 1 \ : \ af(\frac{n}{b}) \leqslant c'f(n) \implies T(n) = \Theta(f(n))$

23-02-23

Fibonacci numbers

In recurrence method, time complexity is $\Omega((\frac{1+\sqrt{5}}{2}^n))$.

If we save values $F_0, F_1, \ldots, F_k$, we get time complexity $\Theta(n)$.

Matrix representation:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

Maximum-Subarray Problem

Given a sequence of numbers, find a contiguous subsequence of numbers which sum is maximized.

We can divide a sequence `A` into 2 pieces, find answer in `A[low...mid)`, then in `A[mid...high)`, then check if the crossing sum is better.

```cpp
#include <iostream>
#include <vector>
#include <limits>

int maxCrossingSubarray(const std::vector<int>& nums, int low, int mid, int high) {
    int leftSum = std::numeric_limits<int>::min();
    int sum = 0;

    // Find the maximum sum of the left subarray
    for (int i = mid; i >= low; --i) {
        sum += nums[i];
        if (sum > leftSum) {
            leftSum = sum;
        }
    }

    int rightSum = std::numeric_limits<int>::min();
```

```
    sum = 0;
    // Find the maximum sum of the right subarray
    for (int i = mid + 1; i <= high; ++i) {
        sum += nums[i];
        if (sum > rightSum) {
            rightSum = sum;
        }
    }

    // Return the sum of the left and right subarrays
    return leftSum + rightSum;
}

int maximumSubarraySum(const std::vector<int>& nums, int low, int high) {
    if (low + 1 <= high) {
        return nums[low];
    }

    int mid = (low + high) / 2;

    int leftSum = maximumSubarraySum(nums, low, mid);
    int rightSum = maximumSubarraySum(nums, mid, high);
    int crossingSum = maxCrossingSubarray(nums, low, mid, high);

    return std::max({leftSum, rightSum, crossingSum});
}

maximumSubarraySum(data, 0, data.size());
```

Time complexity is $O(n \log n)$, since we have an recurrence $T(n) = 2T(\frac{n}{2}) + O(n)$.

Matrix Multiplication

Given matrices A, B, find the result of $C = AB$.

Standard algorithm -- $O(n^3)$:

$A \in R_{n \times m}, \ \ B \in R_{m \times q}$

```
for (int i = 0; i < n; ++i)
    for (int j = 0; i < q; ++j) {
        c[i][j] = 0;
        for (int k = 0; k < m; ++k)
            c[i][j] += a[i][k] * b[k][j];
    }
```

Divide & Conquer algorithm -- $O(n^2)$:

Idea: $n \times n$ matrix $= 2 \times 2$ matrix of $(n/2) \times (n/2)$ submatrices:

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$C = A \cdot B$$

Combining subproblem solutions:

$$\left.\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dg \\ u &= cf + dh \end{aligned}\right\} \begin{aligned} &8 \text{ mults of } (n/2)\times(n/2) \text{ submatrices} \quad \Rightarrow 8T(n/2) \\ &4 \text{ adds of } (n/2)\times(n/2) \text{ submatrices} \quad \Rightarrow 4n^2 \end{aligned}$$

Time complexity is $O(n^2)$, since we have a recurrence $T(n) = 8T(\frac{n}{2}) + 4n^2$

Strassen's Algorithm -- $O(n^{\log 7})$:

Multiply matrices with 7 multiplications and 18 additions.

Need more explanation, yes.

23-02-28

Binary heap

A binary heap is stored in the array. `a[1]` is the root.

```
1 int parent(int i) { return floor(i / 2); }
2 int left(int i) { return 2 * i; }
3 int right (int i) { return 2 * i + 1; }
```

In the root, the min or the max value is stored: $\forall$`i : a[parent(i)]` $\geqslant$ `a[i]`.

The height of the heap is $\log n$.

Proof:

Assume, levels $0, \dots, h-1$ are complete, so there are at least $1 + 2 + \dots + 2^{h-1} = 2^h - 1$ numbers. There is h level, so $n > 2^h - 1 \Rightarrow n \geqslant 2^h$.

If all $0, \dots, h$ levels are complete, the amount of numbers will be $2^{h+1} - 1$.

$2^h \leqslant n \leqslant 2^{h+1} - 1 < 2^{h+1} \;\Rightarrow\; h \leqslant \log n < h + 1.$

$h \in N \;\Rightarrow h = lower\_bound(\log n).$

```
void maxHeapify(vector<int> &a, int n, int i) { // O(logn)
    int l = left(i);
    int r = right(i);
    int largest = i;
    if (l < n && a[l] > a[largest])
        largest = l;
    else if (r < n && a[r] > a[largest])
        largest = r;
    if (largest != i) {
        swap(a[largest], a[i]);
        maxHeapify(a, n, largest);
    }
}

// идем снизу вверх.
// элементы ceil(n/2)+1,...,n всегда являются листьями
build(vector<int> &a, int n) { // O(n)
    for (int i = floor(n/2); i >= 1; --i)
        maxHeapify(a, n, i);
}
```

23-03-06

Decision tree

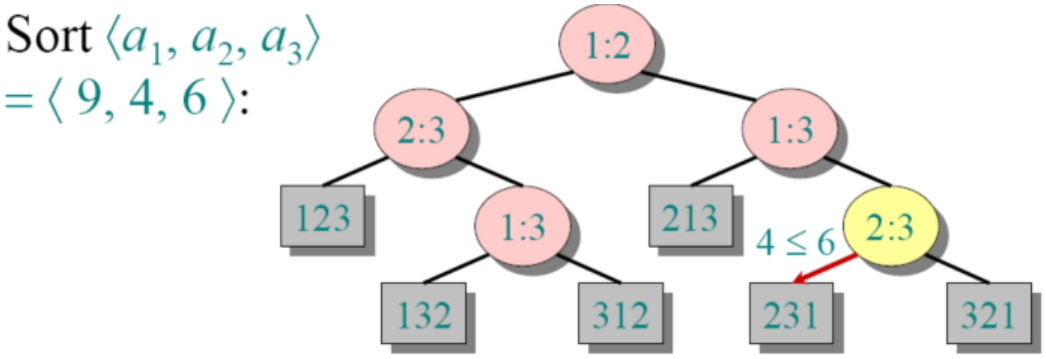
There are nodes with some conditions and leaves. In leaves, divided objects are stored.

Example:

We have a permutation $< a_1, a_2, a_3 >$.

Lets consider some node with numbers i, j. permutations where $a_i < a_j$ go to the left from the node, others go to the right.

$< 9, 4, 6 >$. $a_1 = 9 < a_2 = 4$? No $\Rightarrow$ go to the right from the root.



Any decision tree that can sort n elements should have height $\Omega(n \log n)$.

Proof:

A tree must contain $\geqslant n!$ leaves, since there are $n!$ possible permutation. If the tree height is h, then the tree has $\leqslant 2^h$ leaves. Thus,

$$2^h \geqslant n!$$
$$h \geqslant \log n! \geqslant \log(\frac{n}{e})^n$$
$$h \geqslant n \log n - n \log e$$
$$h = \Omega(n \log n)$$

23-03-10

Binary search

Idea: Use a Divide & Conquer strategy.

1. Divide: Check middle element.
2. Conquer: Recursively search one subarray.
3. Combine: Nothing to be done.

```
int binary_search(vector<int>& a, int x) {
    int l = 0;
    int r = a.size();
    while (r - l > 1) {
        int mid = l + (r - l) / 2;
        if (a[mid] == x) {
```

```
            return mid;
        } else if (a[mid] < x) {
            l = mid;
        } else {
            r = mid;
        }
    }
    return -1;
}
```

Time complexity is $O(\log n)$.

Binary search tree

A *binary search tree* (BST) -- a binary tree with the following property:

For the node `x`, a node `y` is in the left subtree if `y.key` $\leqslant$ `x.key`. Otherwise, it is in the right subtree.

Operations on BST work in $O(\log n)$.

```
struct Node {
    Node* left;
    Node* right;
    Node* parent;
    int x;
}

bool find(Node* v, int x) {
    if (!v->left && !v->right)
        return false;
    if (v->x == x)
        return true;
    if (v-> x > x)
        find(c->right, x);
    find(v->left, x);
}

Node* add(Node* v, int x) {
    if (!v->left && !v->right)
        return new Node(x);
    else if (x < v->x)
        v->left = add(v->left, x);
    else if (x > v->x)
        v->right = add(v->right, x);
}
```

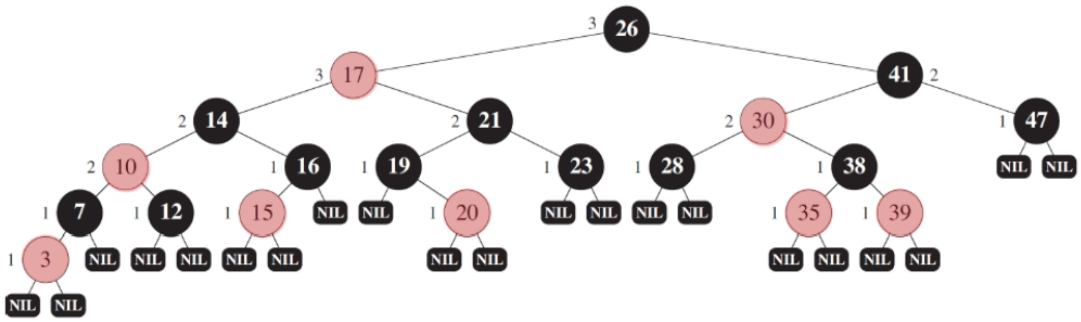
```
26
27  Node* next(Node* v) {
28      if (v->right) {
29          v = v->right;
30          while (v->left)
31              v = v->left;
32      }
33      return v;
34  }
35
36  void del(Node* v) {
37      Node* u = next(v);
38      v = u;
39      u->parent->left = u->right;
40      u->parent = v->parent;
41      u->left = v->left;
42      u->right = v->right;
43  }
44
45  void print(Node*v) {
46      if (!v->right && !v.left)
47          return;
48      print(v->left);     // выведется в отсорт.порядке:
49      std::cout << v->x;  // сначала все <x, потом x, потом >x
50      print(v->right);
51  }
52
53  Node* min(Node* v) {
54      while (v->left)
55          v = v->left;
56      return v;
57  }
58
59  Node* max(Node* v) {
60      while (v->right)
61          v = v->right;
62      return v;
63  }
```

Red-black tree

A *red-black tree* is a BST that also holds the attribute of a color (red or black), which is encoded in one additional bit.

The tree is approximately balanced, since no path from the root to a leaf is more than twice as long as any other path.

- every node is either red or black
- the root is black
- all leaves have `Nil` as key and they are black
- if a node is red, then both children are black
- for each node all paths from the node to a leaf have the same number of black nodes



For each node `x`, we can define a unique black height `bh(x)`.

Lemma (amount of non-leaf nodes)

Let $n(x)$ be the number of non-leaf nodes of a red-black subtree rooted at x. Then, $n(x) \geq 2^{bh(x)} - 1$.

Proof (by induction on height $h(x)$ of node x):

- $h(x) = 0$: x is a leaf. $bh(x) = 0$. $2^{bh(x)} - 1 = 0$. $n(x) \geq 0$. True.
- $h(x) > 0$: x is a non-leaf node. It has two children c_1 and c_2. If c_i is red, then $bh(c_i) = bh(x)$, else $bh(c_i) = bh(x) - 1$. Use assumption, since $h(c_i) < h(x)$, $n(c_i) \geq 2^{bh(c_i)} - 1 \geq 2^{bh(x)-1} - 1$. Thus, $n(x) = n(c_1) + n(c_2) + 1 \geq 2(2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$.

Lemma (comparing `bh` and height)

Let h be the height of a red-black tree with root r. Then, $bh(r) \geq h/2$.
Proof:

- Let $r, v_1, v_2, ..., v_h$ be the longest path in the tree.
- The number of black nodes in the path is $bh(r)$.
- Thus, the number of red nodes is $h - bh(r)$.
- Since v_h is black (LeaB property) and every red node in the path must be followed by a black one (BredB property), we have $h - bh(r) \leq bh(r)$.
- Hence, $bh(r) \geq h/2$.

Th (about height of red-black tree)

A red-black tree with n non-leaf nodes has height $h \leq 2\lg(n+1)$.
Proof:

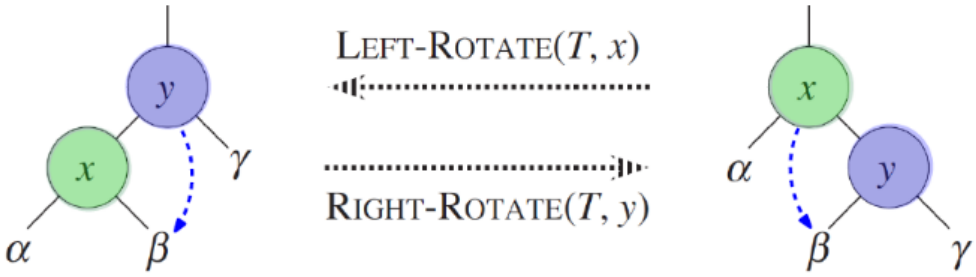
- Lemma 1: $n \geq 2^{bh(r)} - 1$ (r being the root).
- Lemma 2: $bh(r) \geq h/2$.
- Thus, $n \geq 2^{h/2} - 1$.
- So, $h \leq 2\lg(n+1)$.

Note: thus, the height of the red-black tree is $O(\log n)$. All operations can be performed in $O(\log n)$.

Operations with red-black tree

Rotations

- *Right-Rotate*(T, y):
  - Node y becomes right child of its left child x
  - New left child of y is former right child of x
- *Left-Rotate*(T, x):
  - Node x becomes left child of its right child y
  - New right child of x is former left child of y



23-05-19

BFS

V is for a set of vertexes, E is for a set of edges.

```
void bfs(int start, vector<int> &parent, vector<int> &dist) {
    queue<int> q;
    vector<int> used;
    used[start] = true;
    dist[start] = 0;
    q.push(start);
    while (!q.empty()) {
        int v = q.front();
        q.pop();
        for (int &x : g[v]) {
            if (!used[x]) {
                used[x] = true;
                dist[x] = dist[v] + 1;
                parent[x] = v;
                q.push(x);
            }
        }
    }
}
```

Time complexity is $O(V + E)$. We visit every node and go through every edge.

DFS

```
void dfs(v) {
    if (used[v]) {
        return;
    }
    used[v] = true;
    for (int &x : g[v]) {
        dfs(x);
    }
}
```

Time complexity is $O(V + E)$, since we visit every node and go through every edge.