DSA Notes

Tomislav Karastojković, www.alepho.com

1 Divide and Conquer

The technique divides a problem into a set of independent subproblems of the same kind. Each of them is solved and they are merged into a solution. For such approach, the following theorem can be used to compute the complexity.

Theorem 1. Let $a \geq 1$ and $b > 1$ be the constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the reccurence

$$T(n) = aT(\frac{n}{b}) + f(n)$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$
2. If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$
3. If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$ and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta\left(f(n)\right)$

1.1 Towers of Hanoi

Given a three rods and set of $n \geq 1$ disks of different sizes on the first rod, move them all to the third rod by applying the following rules:

1. At each moment, a disk can be on top only of a bigger disk.
2. Only one disk can be moved at a time.
3. Only disk from the top of a rod can be moved.

The algorithm is trivial if number of disks is 1 as well if it's equal to 2. For $n = 3$, the problem can be recursively solved for size of 2 for the middle rod, then third disk is moved to the third rod and problem is again recursively solved for the disks at the middle rod. Similarly, for $n > 3$, the middle rod can be used as auxiliary to solve the subproblem of size $n - 1$. Then n-th disk is moved to the third rod and $n - 1$ disks from the middle rod are moved to the third one by using the same recursive solution for the subproblem of size $n - 1$.

Number of moves in such solution is $2^n - 1$, where n is number of disks. That can be easily prooved by using induction by n.

Algorithm 1 Towers of Hanoi

Input

n Number of tiles.
s Source where to begin moving.
t Target where to end moving.

Output

Tiles moved from the source stick to the target.

Complexity

$O(n^2)$.

procedure HANOITOWERS(n, s, t)
  if $n = 1$ **then**
    Move(s, t)
  else if $n = 2$ **then**
    $a =$ Third(s, t)
    Move(s, a)
    Move(s, t)
    Move(a, t)
  else if $n \geq 3$ **then**
    $a =$ Third(s, t)
    Solve$(n - 1, s, a)$
    Move(s, t)
    Solve$(n - 1, a, t)$
procedure THIRD(s, t)
  if $s \neq 1$ **and** $t \neq 1$ **then return** 1
  if $s \neq 2$ **and** $t \neq 2$ **then return** 2
  if $s \neq 3$ **and** $t \neq 3$ **then return** 3

2 Dynamic programming

Solution of the dynamic problem must be parametrized in a such way that subproblems are of the same kind. That is, if S is an optimal solution for the problem with certain parameters (i.e value V of the problem is optimal), then the subproblem S' obtained from S by removing some element of the problem (and hence by reducing some solution's parameter) is optimal too with the same set of parameters. Parameters by which the problem is expressed and optimization performed is problem *dimension*. Is the dimension unique for the problem?

Constructing hierarchy of subproblems usually involves some ordering or sorting of the problem's elements. When the problem is split into two subproblems, those subproblems are independent i.e. solution of the one has no affect on the solution of the other. The optimal solution is reccurent formula by value of the solution, so using S in that formula isn't possible. We use S for proving an optimal substructure, but expressing formula is always in terms of the V. S and V are parametrized with same parameters, and V can use only problem's elements as parameters.

If we take specific element of the problem (first, last etc.) then we can't assume that this element has some special position in the optimal solution. Conversely, if we take some specific element from the optimal solution, then it can be any element from the problem, i.e. no assumption about it's special position in the solution can be made. There's no sense to make optimization by some parameter. If V is an optimal value, then it can't be parameter of S because S and V have same parameters.

We can try split the optimal solution by removing last element (if that can be done – longest common sequence, optimal edit transformation, knapsack problem where each item is unique). If that isn't possible, we can try by dividing half (matrix-chain multiplication, knapsack problem where number of each item is infinite, maximal set of mutual compatible activities).

To calculate the solution we often need to define the space of subproblems. That is, a set of all subproblems of the given problem, and it's a little bit more complicated than the solution. Usually, space of subproblems is deduced from the recurrent formula of the solution. It is used to calculate optimal solution in bottom-up manner.

Optimal solution of each problem has its own structure, which can contain items, pairs of numbers, stations, etc. To prove optimality it is important to find best structure of the optimal solution which is suited for the proof of optimality. When reccurent formula is given, it is based on the optimal solution, and it's structure is used to express that formula.

2.1 Knapsack with infinite number of items

Knapsack of capacity C should be filled with items $x_1, \dots, x_n$. Items have capacity c_k and value v_k, $1 \le k \le n$, each item can be taken infinite number of times.

Let S be an optimal solution for items $x_1, \dots, x_n$ and capacity C, let $x_k \in S$ for some $k = 1, \dots, n$. Then $S \setminus x_k$ is an optimal solution for $x_1, \dots, x_n, C - c_k$. Thus, S depends on one parameter – capacity of the knapsack. So, $S = S(C)$ and

$$S(C) = S(C - c_k) \cup x_k \text{ for some } k = 1, \dots, n \Rightarrow$$

$$V(C) = \max\{V(C - c_k) + v_k \colon 1 \le k \le n\}$$

Space of subproblems would be $\{S(M) \colon 1 \le M \le C\}$.

Iterative solution uses the same logic but with another loop instead the recursive calls.

2.1.1 Alternative approach

The approach above although correct, lacks of understanding about the problem dimension.

Let S be an optimal solution for items $x_1, \dots, x_n$ and capacity C. If $x_n \in S$, then $S \setminus x_n$ is optimal for $x_1, \dots, x_n$, and $C - c_n$. If $x_n \notin S$, then $S \setminus x_n$ is optimal for $x_1, \dots, x_{n-1}$ and C. Thus, S depends on two parameters – number of items and capacity. So, $S = S(n, C)$ and

$$S(n, C) = \begin{cases} S(n, C - c_n) \cup x_n, & x_n \in S \\ S(n-1, C), & x_n \notin S \end{cases} \Rightarrow$$

Algorithm 2 Knapsack with infinite items, the recursive solution

Input

C Knapsack capacity.

n Number of items.

$i[]$ Items with capacities $i[k].c$ and values $i[k].v$, $k = 1, \ldots, n$.

Output

V, T Optimal solutions of subproblems.

Complexity

$O(n \cdot C)$.

procedure KnapsackInfItemsRec(C)
    $m := 0$
    for $k := 1$ **to** n **do**
        if $V[C - i[k].c] = 0$ **then**
            $V[C - i[k].c] :=$ KnapsackInfItemsRec($C - i[k].c$)
        if $m < V[C - i[k].c] + i[k].v$ **then**
            $m := V[C - i[k].c] + i[k].v$
            $T[C] := k$
    $V[C] := m$
    return m

Read(C, n) ▷ Knapsack capacity and number of items.
Read($i[1 \mathinner{..} n]$) ▷ Items capacities and values.
new $V[1 \mathinner{..} C] := [0 \mathinner{..} 0]$ ▷ Indexed by a subproblem capacity, an optimal solution is determined by stored items indexes of all subproblems from capacity zero up to the given one.
new $T[1 \mathinner{..} C] := [0 \mathinner{..} 0]$
KnapsackInfItemsRec(C)
Write(T, V)

Algorithm 3 Knapsack with infinite items, the iterative solution

Input

C Knapsack capacity.

n Number of items.

$i[]$ Items with capacities $i[k].c$ and values $i[k].v$, $k = 1, \ldots, n$.

Output

V, T Optimal solutions of subproblems.

Complexity

$O(n \cdot C)$.

```
procedure KnapsackInfItemsIter(C)
    for c := 1 to C do
        for k := 1 to n do
            if V[c] < V[c − i[k].c] + i[k].v then
                V[c] := V[c − i[k].c] + i[k].v
                T[C] := k
    return V[C]

Read(C, n) ▷ Knapsack capacity and number of items.
Read(i[1 .. n]) ▷ Items capacities and values.
new V[1 .. C] := [0 .. 0] ▷ Indexed by a subproblem capacity, an optimal solution is determined by
stored items indexes of all subproblems from capacity zero up to the given one.
new T[1 .. C] := [0 .. 0]
KnapsackInfItemsRec(C)
Write(T, V)
```

$$V(n, C) = \max\{V(n, C - c_n) + v_n, V(n-1, C)\}$$

2.2 Knapsack with one item of each kind

Knapsack of capacity C should be filled with items $x_1, \ldots, x_n$, items have capacity c_k and value v_k, $1 \le k \le n$, each item can be taken once.

Let S be an optimal solution for items $x_1, \ldots, x_n$ and capacity C. If $x_n \in S$, then $S \setminus x_n$ is optimal for $x_1, \ldots, x_{n-1}$ and $C - c_n$. If $x_n \notin S$, then $S \setminus x_n$ is optimal for $x_1, \ldots, x_{n-1}$ and C. Thus, S depends on two parameters – number of items and capacity. So, $S = S(n, C)$ and

$$S(n, C) = \begin{cases} S(n-1, C-c_n) \cup x_n, & x_n \in S \\ S(n-1, C) & x_n \notin S \end{cases} \Rightarrow$$

$$V(n, C) = \max\{V(n-1, C-c_n) + v_n, V(n-1, C)\}$$

Space of subproblems would be $\{S(k, M) \colon 1 \le k \le n, 1 \le M \le C\}$.

2.3 Knapsack with fixed number of items of each kind

Knapsack of capacity C should be filled with items $x_1, \ldots, x_n$, items have capacity c_k and value v_k, each item can be taken t_k times, $1 \le k \le n$.

Let S be an optimal solution for items $x_1, \ldots, x_n, t_1, \ldots, t_n$, and capacity C. If $x_n \in S, t_n > 0$, then $S \setminus x_n$ is optimal for $x_1, \ldots, x_n, t_1, \ldots, t_{n-1}, t_n - 1, C - c_n$. If $x_n \notin S$ or $t_n = 0$, then $S \setminus x_n$ is optimal for $x_1, \ldots, x_n, t_1, \ldots, t_{n-1}, C$. Thus, S depends on three parameters – number of items, number of specific item and capacity. So, $S = S(n, t_n, C)$ and

$$S(n, t_n, C) = \begin{cases} S(n, t_n - 1, C - c_n) \cup x_n, & x_n \in S \wedge t_n > 0 \\ S(n-1, t_{n-1}, C), & x_n \notin S \vee t_n = 0 \end{cases} \Rightarrow$$

Algorithm 4 Knapsack with infinite items, the alternative recursive solution

Input

C Knapsack capacity.
n Number of items.
$i[]$ Items with capacities $i[k].c$ and values $i[k].v$, $k = 1, \ldots, n$.

Output

V, T Optimal solutions of subproblems.

Complexity

$O(n \cdot C)$.

```
procedure KnapsackInfItemsRec2(k, C)
    m := 0
    if V[k, C − i[k].c] = 0 then
        V[k, C − i[k].c] := KnapsackInfItemsRec2(k, C − i[k].c)
    m := V[k, C − i[k].c] + i[k].v
    T[C] := k
    if V[k − 1, C] = 0 then
        V[k − 1, C] := KnapsackInfItemsRec2(k − 1, C)
    if m < V[k − 1, C] then
        m := V[k − 1, C]
        T[C] := 0
    return m
```

Algorithm 5 Knapsack with one item, the recursive solution

Input

C Knapsack capacity.
k First k of n items.
$i[]$ Items with capacities $i[k].c$ and values $i[k].v$, $k = 1, \ldots, n$.

Output

V Maximal value of items for k items and capacity C.

Complexity

$O(k \cdot C)$.

```
procedure KnapsackOneItemsRec(k, C)
    m1 := 0, m2 := 0
    if V[k − 1, C] = 0 then
        V[k − 1, C] := KnapsackOneItemsRec(k − 1, C)
    m1 := V[k − 1, C] + i[k].v
    if V[k − 1, C − i[k].c] = 0 then
        V[k − 1, C − i[k].c] := KnapsackOneItemsRec(k − 1, C − i[k].c)
    m2 := V[k − 1, C − i[k].c] + i[k].v
    if m1 > m2 then
        V[k, C] := m2
    else
        V[k, C] := m1
    return V[k, C]

Read(C, n) ▷ Knapsack capacity and number of items.
Read(i[1 .. n]) ▷ Items capacities and values.
new V[1 .. n, 1 .. C] := [0 .. 0]
KnapsackOneItemsRec(n, C)
```

$$V(n, t_n, C) = \max\{V(n, t_n - 1, C - c_n) + v_n, V(n-1, t_{n-1}, C)\}$$

Space of subproblems would be $\{S(k, t_k, M)\colon 1 \le k \le n, 1 \le M \le C\}$.

2.4 Matrix chain product

Matrix chain product $A_1, \ldots, A_n$ with dimensions $t_0 \times t_1, t_1 \times t_2, \ldots, t_{n-1} \times t_n$, should be parenthesized in a such way that number of multiplications is minimal.

Suppose we have an optimal solution $S : (A_1 \cdots A_k)(A_{k+1} \cdots A_n)$ for some $k, 1 \le k < n$. Then, $S_1 : A_1 \cdots A_k$ and $S_2 : A_{k+1} \cdots A_n$ are optimal too. Thus, S depends on two parameters – index of the first and last matrix in the product. So, $S = S(1, n)$ and

$$S(1, n) = S(1, k) \cup S(k, n) \text{ for some } k, 1 \le k \le n \Rightarrow$$

$$V(1, n) = \max\{V(1, k) + V(k, n) + t_{k-1}t_k t_{k+1}\colon 1 \le k \le n-1\}$$

Space of subproblems would be $\{S(i, j)\colon 1 \le i \le j \le n\}$.

2.5 Levenshtein distance

Two words A_m and B_n with given number of characters have to be transformed into each other using minimal number of edit operations: inserting, deleting or replacing a character. For example, *cat* $\to$ *kat* $\to$ *kate* $\to$ *ate* is transformed by replacing c with k, inserting e and deleting k. *Levenshtein* distance is defined as minimum number of such edit operations.

Let $S\colon A_m \to B_n$ be an optimal sequence of edit operations from the first to the second word. Let o be the last edit operation (i - inserting, d - deleting or r - replacing character with the corresponding costs c_i, c_d, c_r). Then, $S \setminus o$ must be optimal solution too. If $o = i$, then $S \setminus i$ transforms A_m into word B_{n-1}, so $S \setminus o\colon A_m \to B_{n-1}$. $S \setminus d$ transforms A_m to C_{n+1} (from which B_n is obtained by deleting a char). For that reason, $S \setminus d$ transforms A_{m-1} to B_n. Easily, one can see that $S \setminus r\colon A_{m-1} \to B_{n-1}$. Thus, S is depending on two parameters – length of the first and second "subword". So, $S = S(m, n)$ and

$$S(m, n) = \begin{cases} S(m, n-1) + i, & o = i \\ S(m-1, n) + d, & o = d \\ S(m-1, n-1) + r, & o = r \end{cases} \Rightarrow$$

$$V(m, n) = \min\{V(m, n-1) + c_i, V(m-1, n) + c_d, V(m-1, n-1) + c_r\}$$

Space of subproblems would be $\{S(i, j)\colon 1 \le i \le m, 1 \le j \le n\}$.

The algorithm is called *Wagner-Fischer algorithm* and goes by dynamic programming technique.

Theorem 2. Levenshtein distance d is a metric, i.e. for all strings x, y, z the following holds:

1. $d(x, y) \ge 0$
2. $d(x, y) = 0 \Leftrightarrow x = y$
3. $d(x, y) = d(y, x)$
4. $d(x, y) \le d(x, z) + d(z, y)$

Proof. The first property follows from the definition of the edit distance as number of edit operations.

For the second property, let's proove $d(x, y) = 0 \Rightarrow x = y$. Suppose $x \ne y$. Then, there exists an optimal transformation $S\colon x \to y$. From that fact it follows that $d(x, y) \ne 0$ which is contradictory to the assumption. The opposite direction $x = y \Rightarrow d(x, y) = 0$ is trivial to proove.

Third property is pretty obvious: number of optimal transformations $x \to y$ is same to number of optimal transformations $y \to x$ (which are reversed).

To proove the fourth property, suppose that there exists string z such that $d(x, z) + d(z, y) < d(x, y)$. That means that edit operations on the left side of inequality are more optimal than those on the right side of inequality, which is contradiction. QED

Algorithm 6 Levenshtein distance with Wagner-Fischer algorithm

Input
- A First word of the length m.
- B Second word of the length n.

Output
- V Number of edit operations to do.
- S Edit operations to perform.

Complexity
- $O(k \cdot C)$.

```
procedure LEVENSHTEIN(m, n)
    if m = 1 and n = 1 then
        ▷ Transformations of strings of length 1 goes by replacing.
        V[1, 1] := c_r, S[1, 1] := 'r'
        return
    else if m = 1 then
        if V[1, n − 1] = 0 then
            Levenshtein(1, n − 1)
        V[1, n] := V[1, n − 1] + c_d, S[1, n] := 'd'
    else if n = 1 then
        if V[m − 1, 1] = 0 then
            Levenshtein(m − 1, 1)
        V[m, 1] := V[m − 1, 1] + c_i, S[m, 1] := 'i'
    else
        if V[m, n − 1] = 0 then
            Levenshtein(m, n − 1)
        V[m, n] := V[m, n − 1] + c_i, S[m, n] := 'i'
        if V[m, n] > V[m − 1, n] + c_d then
            V[m, n] := V[m − 1, n] + c_d, S[m, n] := 'd'
        if V[m, n] > V[m − 1, n − 1] + c_r then
            V[m, n] := V[m − 1, n − 1] + c[r], S[m, n] := 'r'

Read(c_i, c_d, c_r) ▷ Costs of edit operations.
Read([1 .. m], B[1 .. n]) ▷ Optimal solutions of subproblems.
new V[1 .. m, 1 .. n] := [0..0], S[1 .. m, 1 .. n] := [''..'']
Levenshtein(m, n)
```

2.6 Damerau-Levenshtein distance

Damerau-Levenshtein distance adds transposition of two adjacent characters to Levenshtein distance. Transposition occurs in cases like *deamon* $\to$ *daemon*.

If $S\colon A_m \to B_n$ is an optimal transformation and the last operation is transposition t of cost c_t, then $S \setminus t$ is optimal for $A_{m-2} \to B_{n-2}$. Thus,

$$S(m,n) = \begin{cases} S(m,n-1)+i, & o=i \\ S(m-1,n)+d, & o=d \\ S(m-1,n-1)+r, & o=r \\ S(m-2,n-2)+t, & o=t \end{cases} \Rightarrow$$

$$V(m,n) = \min\{V(m,n-1)+c_i, V(m-1,n)+c_d, V(m-1,n-1)+c_r, V(m-2,n-2)+c_t\}$$

2.7 Longest common sequence

Let $X_m = (x_1,\ldots,x_m), Y = (y_1,\ldots,y_n)$ be two sequences. Find the longest common sequence of X_m and Y_n.

Suppose that S is optimal solution i.e. longest common sequence of X_m and Y_n. If $x_m = y_n$, then $S \setminus x_m$ is optimal for X_{m-1} and Y_{n-1}. If $x_m \neq y_n$, then $S \setminus x_m$ is better solution of the solutions for X_{m-1}, Y_n and X_m, Y_{n-1}, which are optimal too. Thus, S is depending of two parameters – length of the subsequence of X and Y. So, $S = S(m,n)$ and:

$$S(m,n) = \begin{cases} S(m-1,n-1) \cup x_m, & x_m = y_n \\ \text{best of } S(m-1,n), S(m,n-1), & x_m \neq y_n \end{cases} \Rightarrow$$

$$V(m,n) = \begin{cases} V(m-1,n-1)+1, & x_m = y_n \\ \max\{V(m-1,n), V(m,n-1)\}, & x_m \neq y_n \end{cases}$$

Space of subproblems would be $\{S(i,j)\colon 1 \le i \le m, 1 \le j \le n\}$.

2.8 Maximal set of activites

Let $a_1,\ldots,a_n$ be the activites with starting and finishing times, s_k and f_k, $1 \le k \le n$. Find the maximal subset of mutual compatible activites.

Suppose that S is an optimal solution, let $a_k \in S$ be an activity. $S \setminus a_k$ is splitted into two subsets: $S = S_1 \cup S_2$, S_1 - activites that finish before s_k, S_2 - activites that start after f_k. Then, S_1 and S_2 are optimal too. Thus, S is depending of two parameters – starting and finishing time. So, $S = S(0,\infty)$ with activites that start after 0 and finish before ∞ and

$$S(0,\infty) = S(0,s_k) \cup a_k \cup S(f_k,\infty) \text{ for some } a_k = (s_k,f_k), 1 \le k \le n \Rightarrow$$

$$V(0,\infty) = \max\{V(0,s_k)+1+V(f_k,\infty)\colon 1 \le k \le n\}$$

Space of subproblems would be $\{S(i,j)\colon 0 \le i \le j \le \infty\}$.

2.9 Minimal number of halls

Let $a_1,\ldots,a_n$ be the activites with starting and finishing times, s_k and f_k, $1 \le k \le n$. Find the minimal numbers of halls so all activities could be realized.

Optimal solution S is composed of the solutions for single halls, i.e. $S = S_n \cup S_{n-|S_1|} \cup \ldots \cup S_{n-\sum_{k=1}^{n-1}|S_k|}$, where S_k is optimal solution for single hall and k activites. Thus, S is reduced to solve on single halls.

2.10 Machine jobs

Let $x_1, \ldots, x_n$ be jobs to be done on one machine. Each job lasts for t_k time units, must be finished before deadline d_k and makes profit p_k. One job at the time can be done on the machine. Find jobs such that profit is maximal.

We can assume that $d_1 \leq \ldots \leq d_n$; otherwise, we can renumarate jobs so this holds true. Let S be an optimal solution for $x_1, \ldots, x_n, d_n, t_1 + \ldots + t_n$. If $x_n \in S$, then $S \setminus x_n$ is optimal for $x_1, \ldots, x_{n-1}, d_n - t_n, t_1 + \ldots + t_{n-1}$. If $x_n \notin S$, then $S \setminus x_n$ is optimal for $x_1, \ldots, x_{n-1}, d_{n-1}, t_1 + \ldots + t_{n-1}$. Thus, S depends of three parameters – number of jobs, deadline and total time for these jobs. So, $S = S(n, d_n, t_1 + \ldots + t_n)$ and

$$S(n, d_n, t_1 + \ldots + t_n) = \begin{cases} S(n-1, d_n - t_n, t_1 + \ldots + t_{n-1}) \cup x_n, & x_n \in S \\ S(n-1, d_{n-1}, t_1 + \ldots + t_{n-1}), & x_n \notin S \end{cases} \Rightarrow$$

$$V(n, d_n, t_n) = \max\{V(n-1, d_n - t_n, t_1 + \ldots + t_n) + p_n, V(n-1, d_{n-1}, t_1 + \ldots + t_{n-1})\}$$

Space of subproblems would be $\{S(k, d_k, t_1 + \ldots + t_k) \colon 1 \leq k \leq n\}$.

2.11 Stations

Let $s_1, \ldots, s_n$ be stations with distances between them: $d_1 = d(s_2, s_1), \ldots, d_{n-1} = d(s_n, s_{n-1})$. When car's tank is full it has gas for l miles. Find minimal number of stops for the car to traverse the whole road.

Let S be an optimal solution for n stations, distance $d_1 + \ldots d_{n-1}$, tank with gas for l miles. If car stops in s_2, then it has $n-1$ stations, distance $d_2 + \ldots + d_{n-1}$ and l gas. If car does not stop in s_2, then it has $n-1$ stations, distance $d_2 + \ldots + d_{n-1}$ and $l - d_1$ gas. Solution $S \setminus s_1$ is optimal for $s_2, \ldots, s_n, d_2 + \ldots + d_n, l$ or $l - d_1$ (depending of that if the car has stopped). Thus, S is depending of three parameters – number of stations, distance to traverse and gas in the tank. So, $S = S(n, d_1 + \ldots + d_{n-1}, l)$ and

$$S(n, d_1 + \ldots + d_{n-1}, l) = \begin{cases} S(n-1, d_2 + \ldots + d_{n-2}, l), & \text{stopped at } s_2 \\ S(n-1, d_2 + \ldots + d_{n-2}, l - d_1), & \text{not stopped at } s_2 \end{cases} \Rightarrow$$

$$V(n, d_1 + \ldots + d_{n-1}, l) = \min\{V(n-1, d_2 + \ldots + d_{n-2}, l) + 1, V(n-1, d_2 + \ldots + d_{n-2}, l - d_1)\}$$

Let's proove that dynamic solution can be transformed into greedy solution. Let s_k be the station where car has stopped when no gas is available to reach next station, i.e. $k \leq n$ is such that $d_1 + \ldots + d_{k-1} \leq l < d_1 + \ldots + d_k$. Then, s_k is the first station to stop for the car. To prove this we need to prove:

1. s_k belongs to some optimal solution
2. $V(n, d_1 + \ldots + d_{n-1}, l) = V(n-k, d_{n-k+1}, l - (d_1 + \ldots + d_{k-1})) + 1$

Proof for 2. is trivial because s_k is chosen such that reccurent formula looses *min* function and all subproblems but one. Let's prove 1. Suppose that S is an optimal solution which doesn't contain s_k. Then, $s_l, l < k$, must belong to S ($l > k$ is not possible). Thus, $S \setminus s_l \cup s_k$ is also an optimal solution which contradicts to the assumption that $s_k \notin S$.

2.12 Greedy solution of the fractional knapsack problem

Proove that fractional knapsack problem has the greedy-choice property.

Let $a_1, \ldots, a_n$ be items sorted in descending order of $\frac{v_k}{w_k}, 1 \leq k \leq n$, where v_k is value, w_k is weight of the k-th item. If optimal solution S does not contain fraction $f_1 \leq w_1$ of the item a_1, then it contains fraction $f_k = f_1, k > 1$. So, S contains fraction $f_k = f_1, k > 1$, where $\frac{v_k}{w_k} < \frac{v_1}{w_1}$. Then, $S \setminus f_k \cup f_1$ is better solution than S, which is contradiction. Thus, S contains fraction f_1 of the item a_1 with maximal average value $\frac{v_1}{w_1}$.

2.13 Maximal product

Let A, B be two sets with n integers. Reorder those integers so $\prod_{i=1}^{n} a_i^{b_i}$, is maximal, where $a_i \in A, b_i \in B$.

Let S be an optimal solution for n numbers $a_1, \ldots, a_n, b_1, \ldots, b_n$. Then, $S \setminus (a_1, b_1)$ is optimal for some a_1, b_1. Thus, $S = S(n)$ and

$$S(n) = (a_1, b_1) \cup S(n-1) \text{ for some } a_1 \in A, b_1 \in B \Rightarrow$$

$$V(n) = \max\{a^b V(n-1) \colon a \in A, b \in B\}$$

Let's prove that this solution can be transformed to a greedy solution, when $a_1 = \max A, b_1 = \max B$. To prove this we need to prove:

1. $a_1 = \max A, b_1 = \max B$ belong to some optimal solution
2. $V(n) = a^b V(n-1)$

Proof for 2. is trivial because a, b are chosen such that formula looses max function. Suppose that S is an optimal solution where $a_1 \neq \max A$ or $b_1 \neq \max B$. Then, S is solution such that $\prod_{k=1}^{n} a_k^{b_k}, a_1 \neq \max B, b_1 \neq \max B, k = 1, \ldots, n$. Let be $a = \max A, b = \max B$. Then,

$$\prod_{k=1}^{n} a_k^{b_k} < a^b \prod_{k=1, a_k \neq a, b_k \neq b}^{n} a_k^{b_k}$$

from which follows that S is not optimal because we've found better reordering of A, B. This is a contradiction.

3 Backtracking

The technique involves examining of all possible solutions and quitting a search as soon as it shows that it does not lead to a solution.

3.1 N queens problem

Problem: On a chess board of size $n \times n$, one should place n queens, so no two queens threaten to each other.

The solution starts from the first column, by placing a queen Q at the first row. Then it tries to place queens at the subsequent columns by verifying that there are no other queens threaten to Q placed at previous columns. To perform such check, the function CheckCell$(row, column)$ assumes that queens are placed on the previous columns. So, it verifies that for a queen Q at the field $(row, column)$ there are no other queens at the same row, upper left or bottom left part of diagonals.

The main function is QueensSolve$(column)$ which checks whether a queen can be placed at the given column. If there are no threats by queens at the previous columns (by recursively calling QueensSolve), the board field is updated to true and the procedure recursively proceeds with the subsequent columns. If the recursion gives a negative answer, then the field is updated to false and the backtrack is performed by repeating the algorithm with other positions as possible solutions.

3.2 Knight's tour

Problem: On a chess board of size 8, starting from the field $(1, 1)$, a knight should visit each field exactly once.

The solution starts from the field $(1, 1)$ and checks for available fields. If not visited, then the algorithm marks it as visited and proceeds recursively. If there are no available fields, then the current field does lead to the solution and it's marked as not visited and the algorithm proceeds with other fields.

3.3 Sudoku

Problem: Given a grid 9×9 with partially filled cells with digits $1, \ldots, 9$ (empty cell has zero) fill all other cells with the digits so the following conditions are met:

1. Each digit is unique in each row and each column.
2. Each digit is unique in each of nine subgrids of size 3 x 3.

The algorithm would be:

1. Go cell by cell from $(1, 1)$ to $(9, 9)$ and for each such cell check:
   (a) is the digit unique in the row/column
   (b) does the digit fit to a subgrid
2. If the check is positive, repeat the algorithm recursively on the next cell.
3. If the check is negative, try another digit with the current cell. If such digit is not available then go back one cell and repeat the check with next available digit.
4. If all cells are populated, a solution is found.

3.4 Longest possible route

Problem: Given a matrix $m \times n$ with a few hurdles arbitrary placed, find the longest possible route from a source to a destination field. Visiting goes by moving to adjacent cells which are not hurdles. Diagonal moves are not allowed. Visiting the same cell again on a path is not allowed.

Algorithm 7 N queens on a chess board

```
Input
    r     Cell's row where to put the queen.
    c     Cell's column where to put the queen.
Output
          Returns true if a queen can be placed at the given position, false if not.
procedure CheckCell(r, c)
    ▷ Check columns at the same row before this one.
    for 1 to c − 1 do
        if B[r][i] = true then
            return false
    ▷ Check upper left part of the diagonal.
    for i := r − 1 downto 1, j := c − 1 downto 1 do
        if B[i][j] = true then
            return false
    ▷ Check lower left part of the diagonal.
    for i := r + 1 to n, j := c − 1 downto 1 do
        if B[i][j] = true then
            return false
    return true

Input
    c     Column to check whether a queen can be placed.
Output
    B     Cells set to true if the queen can be placed at some row at the column c.
          Returns true if the queen can be placed at the column c.
procedure QueensSolve(c)
    ▷ Recursion end.
    if c > n then
        return true
    ▷ Traverse rows of the given column.
    for i := 1 to n do
        if CheckCell(i, c) = true then
            B[i][c] := true
            if QueensSolve(c + 1) = true then
                return true
            ▷ Current field does not lead to the solution, do the backtrack.
            B[i][c] := false
    return false

▷ Matrix with cells holding true where the queen can be placed.
new B[1 .. n][1 .. n]
for i := 1 to n do
    for j := 1 to n do
        B[i][j] := false
QueensSolve(1)
```

Algorithm 8 Knight's tour

Input

r Row to check whether it is part of the chess board.

c Column to check whether it is part of the chess board.

Output

True if does, false if not.

```
procedure CheckField(r, c)
    return 1 ≤ r ≤ 8 and 1 ≤ c ≤ 8
```

Input

r Row to check whether it is part of the solution.

c Column to check whether it is part of the solution.

Output

B Matrix with fields enumerated as the knight visits them, zero if not visited yet.

```
procedure KnightSolve(r, c)
    if B[r][c] = 0 then
        B[r][c] := g
        g := g + 1
    if g > 64 then
        return true
    for j in J do
        ▷ Next row and column.
        r′ := r + j[1], c′ := c + j[2]
        if CheckField(r′, c′) = true and B[r′][c′] = 0 then
            return KnightSolve(r′, c′)
    ▷ No jump leads to the solution, do the backtrack.
    B[r][c] := 0
    g := g − 1
    return false

J := {(−1, +2), (+1, +2), (+2, +1), (+2. − 1), (−1, −2), (+1, −2), (−2, +1), (−2, −1)} ▷ Jumps.
g := 1 ▷ Jumps counter.
new B[1 .. 8][1 .. 8] ▷ Chess board.
for i := 1 to 8 do
    for j := 1 to 8 do
        B[i][j] := 0
KnightSolve(1, 1)
```

Algorithm 9 Sudoku

Input
r Row where to check whether the digit has a conflict.
c Column where to check whether the digit has a conflict.
d Digit to check whether it conflicts to the same digit in the grid.

Output
True if does, false if not.

```
procedure CheckField(r, c, d)
    if G[r, c] ≠ 0 then
        return false
    ▷ Check the row.
    for i := 1 to 9 do
        if i ≠ c and G[r, i] = d then
            return false
    ▷ Check the column.
    for i := 1 to 9 do
        if i ≠ r and G[i, c] = d then
            return false
    ▷ Check the subgrid.
    g_x := (r − 1)/3, g_y := (c − 1)/3
    for i := 3 · g_x + 1 to 3 · g_x + 3 do
        for j := 3 · g_y + 1 to 3 · g_y + 3 do
            if (i, j) ≠ (r, c) and G[i, j] = d then
                return false
    return true
```

Input
r Row where to look for the next cell.
c Column where to look for the next cell.

Output
Next cell if available, null if not.

```
procedure NextCell(r, c)
    if r < 9 then
        return (r + 1, c)
    if r = 9 and c < 9 then
        return (1, c + 1)
```

Algorithm 10 Sudoku

Input
r Row where to examine the grid whether a digit can be placed.
c Column where to examine the grid whether a digit can be placed.

Output
True if does, false if not.

```
procedure SudokuSolve(r, c)
    for i := 1 to 9 do
        for j := 1 to 9 do
            for d := 1 to 9 do
                if G[r, c] = 0 then
                    if CheckField(r, c, d) = true then
                        G[r, c] := d
                        (r', c') := NextField(r, c)
                        if (r', c') = null then
                            return true
                        SudokuSolve(r', c')
```

Input
G Grid partially filled with digits.

Output
Whole grid filled with digits according to the rules.

```
▷ Matrix of digits, partially filled with digits 1-9, or 0 if empty
Read G[1 .. 9, 1 .. 9]
SudokuSolve(1, 1)
```

Algorithm 11 Longest possible route

Input
r Source row where to start the possible route.
c Source column where to start the possible route.
D Destination of the longest route.

Output
Longest possible route.

```
procedure LongestRouteSolve(r, c, D)
    C.push((r, c))
    if (r, c) = D then
        if C.length > M.length then
            M = C
    else
        if G[r − 1, c] = 0 then
            G[r − 1, c] = 1
            C.push((r − 1, c))
            Solve(r − 1, c)
        if G[r + 1, c] = 0 then
            G[r + 1, c] = 1
            C.push((r + 1, c))
            Solve(r + 1, c)
        if G[r, c − 1] = 0 then
            C.push((r, c − 1))
            Solve(r, c − 1)
        if G[r, c + 1] = 0 then
            C.push((r, c + 1))
            Solve(r, c + 1)
```

Input
G Grid is initialized to zero except the cells with hurdles which are -1.

Output
M Maximum path found.

```
Read(G[1 .. m][1 .. n])
new C[] ▷ Current path as list of cells.
new M[]
Read(S, D)
Solve(S.r, S.c)
```

4 Sorting arrays

Common problem is to sort an array of n elements. In this section, array indexes go $1 \dots n$.

4.1 All Pairs Sort

The naive approach compares all elements of an array. Beside an array, a single linked list can be sorted this way.

Algorithm 12 All Pairs Sort

Input
A Array to sort of length n.
Output
A Sorted array in the increasing order.
Complexity
$O(n^2)$.

```
procedure AllPairsSort(A)
    for i := 1 to n − 1 do
        for j := 2 to n do
            if A[i] > A[j] then
                Swap(A[i], A[j])
```

It can be improved by starting index j from i's consecutive.

Algorithm 13 All Pairs Sort Fast

Input
A Array to sort of length n.
Output
A Sorted array in the increasing order.
Complexity
$O(n^2)$.

```
procedure AllPairsSort(A)
    for i := 1 to n − 1 do
        for j := i + 1 to n do
            if A[i] > A[j] then
                Swap(A[i], A[j])
```

4.2 Selection Sort

It is similar to the All Pairs Sort. The difference is that it memoizes the index of the minimal element so far traversed.

4.3 Insertion Sort

It inserts a new coming element into an already sorted subarray. The insertion sort algorithm could be used for double linked lists.

4.4 Bubble Sort

It repeatedly goes through the array and compares the current element with the one after it.

Algorithm 14 Selection Sort

Input
  A Array to sort of length n.
Output
  A Sorted array in the increasing order.
Complexity
  $O(n^2)$.

```
procedure SelectionSort(A)
    for i := 1 to n do
        m := i ▷ Minimal element found so far.
        for j := i + 1 to n do
            if A[j] < A[m] then
                m := j
                if m ≠ i then
                    Swap(i, m)
```

Algorithm 15 Insertion Sort

Input
  A Array to sort of length n.
Output
  A Sorted array in the increasing order.
Complexity
  $O(n^2)$.

```
procedure InsertionSort(A)
    for i := 2 to n do
        v := A[i]
        j := i
        ▷ Insert v into already sorted A[1 .. j].
        while j > 1 and A[j] > v do
            A[j + 1] := A[j]
            j := j − 1
        A[j + 1] := v
```

Algorithm 16 Bubble Sort

Input
  A Array to sort of length n.
Output
  A Sorted array in the increasing order.
Complexity
  $O(n^2)$.

```
procedure BubbleSort(A)
    repeat
        s := false ▷ Is at least one swap made.
        for i := 1 to n − 1 do
            if A[i − 1] > A[i] then
                Swap(A[i − 1], A[i])
                s := true
    until not s
```

4.5 Merge Sort

Divides an array into two subarrays. Each of the subarrays are divided further until one element remains which is trivially sorted. The subarrays are repeatedly merged into the sorted array.

4.6 Heap Sort

Sorting can be done by making max heap, placing first element of the heap at the end of array and decreasing heap. Total runtime is $O(n)$ for MakeHeap and $n/2$ calls for DownHeap which has complexity $O(\lg n)$ so the total runtime is

$$O(n) + \frac{n}{2}O(\lg n) = O(n) + O(n \lg n) = O(n \lg n)$$

4.7 Quick Sort

The quick sort algorithm picks an element from the array A, so called pivot. Then it divides the array so that all elements to the left of the pivot are smaller and all elements to the right of the pivot are greater. The procedure is repeatedly called until all elements are sorted.

The pivot element can be arbitrary taken, in this case it is set to the begin of the array. It is exchanged with the other elements, so that it partitions the array.

4.8 Stack Sort

By using two stacks, S with elements and another T as auxiliary, the sort can be achieved like this:

1. pop an element x from S while the top element on T is bigger than x
2. pop all elements from T and push them to T
3. push x to T
4. repeat the steps above until T is non-empty

Complexity of the algorithm is $O(n^2)$.

4.9 Selection

Let A be a set of n elements. For an integer $1 \leq k \leq n$, find the element $a \in A$ such that a is larger than exactly $k - 1$ elements.

For the solution, the Partition algorithm from Quick Sort is used.

Algorithm 17 Merge Sort

Input

A Array to sort of length n.

Output

A Sorted array in the increasing order.

Complexity

$O(n \log(n))$.

```
procedure MergeSort(A, l, r)
    ▷ Case with only one element.
    if r := l + 1 then return
    ▷ Case with two elements.
    if r := l + 2 then
        if A[l] > A[r] then
            Swap(A[l], A[r])
        return
    ▷ General case.
    m := (l + r)/2
    MergeSort(A, l, m)
    MergeSort(A, m + 1, r)
    MergeSort(A, l, m, r)
procedure Merge(A, l, m, r)
    new B[1 .. n] ▷ Auxilliary array to keep sorted subarrays of A
    h1 := l ▷ Traverses first half
    h2 := m + 1 ▷ Traverses second half
    c := 1 ▷ Traverses both halves
    while h1 ≤ m and h2 ≤ r do
        if A[h1] < A[h2] then
            B[c] := A[h1]
            h1 := h1 + 1
        else
            B[c] = A[h2]
            h2 := h2 + 1
        c := c + 1
    while h1 ≤ m do
        B[c] := A[h1]
        h1 := h1 + 1
        c := c + 1
    while h2 ≤ r do
        B[c] := A[h2]
        h2 := h2 + 1
        c := c + 1
    for i := 1 to n do
        A[l + i] := B[i]
```

Algorithm 18 Heap Sort

Input

A Array with n elements.

Output

A Sorted array.

Complexity

$O(n \lg n)$.

```
procedure HeapSort(A)
    MakeMaxHeap(A)
    for k := n/2 to 1 do
        Swap(1, k)
        n := n − 1
        DownMaxHeap(1)
```

Algorithm 19 Quick Sort

Input

A Array to sort of length n.
l Begin of the subarray to sort.
r End of the subarray to sort.

Output

A Sorted array in the increasing order.

Complexity

$O(n \log(n))$.

```
procedure QuickSort(A, l, r)
    if l < r then
        p = Partition(l, r)
        QuickSort(l, p − 1)
        QuickSort(p + 1, r)
```

Input

A Array to partition.
l Begin of the subarray to partition.
r End of the subarray to partition.

Output

Pivot element

```
procedure Partition(A, l, r)
    i := l, j := r
    p := i  ▷ Pivot is set to the begin.
    while i ≤ j do
        while A[i] ≤ A[p] and i ≤ r do
            i := i + 1
        while A[j] > A[p] and j ≥ l do
            j := j − 1
        if i < j then
            Swap(i, j)
    Swap(i, p)
    return i
```

Algorithm 20 Selection

Input

A Array with n elements.

l Index of the element to start selection from.

r Index of the element to stop selection for.

k k-the element to select.

Output

A Index of the k-th element.

Complexity

$O(n \lg n)$.

procedure SELECTION(A, l, r, k)
  if $l < r$ **then**
    $t :=$ Partition($l, t-1, k$)
    if $t > l + k - 1$ **then return** Selection($l, t-1, k$)
    if $t < l + k - 1$ **then return** Selection($t+1, r, k-t$)
  return k

5 Hash Table

Motivation is to have an array-like data structure where keys can be of any data type (string for instance). Operations of interests are searching, inserting and deleting.

5.1 Definition

Let $T[m]$ be an array. If function $h : U \to \{1, \dots, m\}$ is given, then T is *hash table* $T[h(k)]$, with keys $k \in U$. An element with key k *hashes* to hash value $h(k)$. If $h(k) = h(l)$ for two distinct keys $k, l \in U$, then a *collision* is encountered. Since $|U| > m$, there must be at least two keys with a same hash values, so a method for resolving collisions is needed. There are several approaches:

1. chaining
2. open addressing:
   (a) linear probing
   (b) quadratic probing
   (c) double hashing

Chaining keeps elements with the same hash value in a linked list. Inserting a key K is inserting at the head of list $T[h(K)]$. Searching/deleting a key K is searching/deleting it in the list $T[h(K)]$. *Load factor* is $\alpha = n/|T|$, where n is number of elements stored in hash table. If h is simple uniform hash function, then searching can be accomplished in $\Theta(1 + \alpha)$ time. If the hash table size is proportional to the number of elements in the table, then searching is made in $\Theta(1)$. Inserting and deleting take $O(1)$ worst-time when the lists are doubly linked.

Open addressing stores all keys in the array T, i.e. no lists per slot. When slot $h(K)$ is already occupied, hash table is being examined until free slot is found; that is so called *probe sequence*.

Linear probing increases the interval between probes by one. used function is $h(k, i) = (h'(k) + i) \bmod m$

Quadratic probing increases the interval between probes linearly used function is $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

Double hashing uses hash function of the form $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$

5.2 Hash functions

A good hash function should uniform hash it's keys. Most hash functions assume that $U = \mathbb{N}$ because it is often possible to make such transformation. For example, string can be transformed into natural number by summing it's ASCII code characters.

Algorithm 21 Hash

Input
- T Hash table.
- K String to hash.

Output
Hash value of K.

Complexity
$O(n \lg n)$.

procedure HASH(K)
  $v := 0$ ▷ Hash value of K
  for $i := 1 \to |K|$ **do**
    $v := v + K[i]$
  return $v \bmod |T|$

5.3 Division method

Hash function $h : U \to \{0, \dots, m-1\}$ is $h(k) = k \bmod m$.

5.4 Multiplication method

5.5 Open addressing method

6 Binary Heap

Motivation for the binary heap is to have structure which enables fast retrieval of maximum or minimum key, as for instance in the priority queue. This principle is also used by Heap Sort at 4.6. Operations of interest are inserting an element into heap, deleting and getting maximum/minimum element.

6.1 Definition

Definition 6.1. *Min binary heap* is a binary tree containing keys with the following properties:

1. Every level, except possibly the last, is completely filled with nodes and all nodes are as far left as possible.
2. If y is a child node of x, then $x.key \leq y.key$.

For the *max binary heap* the second condition is modified to be $x.key \geq y.key$. These definitions are equivalent to the definitions from the section Tree at 8.1. In this section only max heap is considered, min heap is analogous.

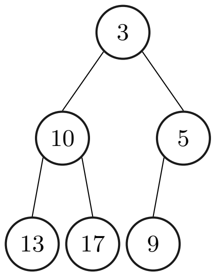

Figure 1: Min binary heap of $n = 7$.

In the pseudo code, the n-sized heap is represented as an n-sized array with indexes $1, 2, \ldots, n$. It's elements are access via index operator $a[k]$. Parent of an element $a[k]$ is determined as $a[k/2]$ and children as $a[2k]$ and $a[2k+1]$.

6.2 Heapifying

Heapifying an element x puts the element at the right place in the heap, so the heap property is maintained. It can be performed in bottom-up or top-down manner.

Bottom-up method assumes that first $k-1$ elements of n-sized array a already form the heap. Then, the k-th element $a[k]$ has to be moved onto the right position so $a[1], \ldots, a[k]$ elements form the heap. The procedure moves $a[k]$ upwards until it reaches element less than $a[k]$.

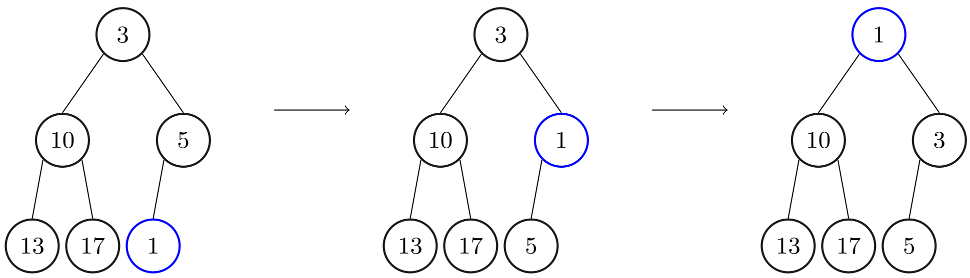

Figure 2: Bottom-up heapifying on the newly added node 1.

Theorem 3. The running time of UpHeap is $O(\log n)$ for an array a of n elements.

Algorithm 22 Heapify the binary heap to top

Input

k-th element to insert into $k-1$-sized heap of n-sized array a.

Output

$a[1], \ldots, a[k]$ forming the heap.

Complexity

$O(\log n)$.

procedure $\text{UpHeap}(k)$
    $v := a[k]$
    while $k/2 > 0$ **and** $v \leq a[k/2]$ **do**
        $a[k] := a[k/2]$
        $k := k/2$
        $a[k] := v$

Proof. The running time at k-th element is $O(1)$ for fixing relationships among $a[k]$ and it's parent plus the time for all it's parent nodes up to the top. Maximum size of the subtree above the k-th element is $n/2$ so the reccurence is

$$F(n) = F(\frac{n}{2}) + \Theta(1)$$

which solution by master theorem case 2 is $F(n) = O(\log n)$. QED

Top-down method observes $a[k+1], \ldots, a[n]$ of n-sized array a as leaves of the heap. The algorithm heapifies $a[k], \ldots, a[n]$ by putting k-th element at the proper position.

Algorithm 23 Heapify the binary heap to bottom

Input

k-th element to insert into heap $a[k+1], \ldots, a[n]$ of n-sized array a.

Output

Heap $a[k], \ldots, a[n]$ at n-sized array a, $a[1], \ldots a[k-1]$ not part of the heap.

Complexity

$O(\log n)$.

procedure $\text{DownHeap}(k)$
    $v := a[k]$
    while $k < n/2$ **and** $v \leq a[2k]$ **do**
        $j := 2k$
        if $j < n$ **and** $a[j] < a[j+1]$ **then**
            $j = j + 1$
        $a[k] := a[j]$
        $k = j$
        $a[k] := j$

Theorem 4. The running time of DownHeap is $O(\log n)$ for an array a of n elements.

Proof. The running time at k-th element is $O(1)$ for fixing relationships among $a[k]$ and it's children plus the time for a subtree of some of its children. Maximum size of such subtree is $\frac{2n}{3}$, which is case when the last row is half full. Therefore, the reccurence is

$$F(n) \leq F(\frac{2n}{3}) + \Theta(1)$$

which solution by master theorem case 2 is $F(n) = O(\log n)$. QED

6.3 Creating heap

Making new heap of an existing array a with n elements can be imagined as heapifying first $n/2$ elements assuming that second $n/2$ elements already form a heap as it's leaves.

Algorithm 24 Making the binary heap

Input
Array a with n elements.

Output
a as a heap.

Complexity
$O(n)$.

procedure MakeHeap(k)
  for $i := n/2$ **to** 1 **do**
    DownHeap(i)

Theorem 5. The running time of MakeHeap is $O(n)$.

Proof. Since there are $n/2$ operations and each of them has complexity $O(\log n)$ total complexity is $O(n \log n)$. Although this bound is true, it is not asmptotically tight.

Tighter bound can be obtained by observing that an n-element heap has height $\lfloor \log n \rfloor$ and at most $\lceil n/2^{h+1} \rceil$ nodes at any height h. Total time of DownHeap when called on a node of height h is $O(h)$, so the total cost of MakeHeap is

$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil n/2^{h+1} \rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) \leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(2n) = O(n)$$

QED

6.4 Inserting key

Inserting new elements consists of adding new element at the end of the heap and then moving into a correct place. Complexity is actual the complexity of the UpHeap.

Algorithm 25 Insert into the binary heap

Input
K Key insert into heap a of n elements.

Output
$n + 1$-sized heap a.

Complexity
$O(\log n)$.

procedure HeapInsert(K)
  $n := n + 1$
  $a[n] := K$
  UpHeap(n)

6.5 Deleting key

Deleting k-th element is placing the last element on it's place and heapifying subheap with k-th element as root. Complexity is actual complexity of DownHeap.

Algorithm 26 Deleting from the binary heap

Input

k-th element to delete, $1 \leq k \leq n$, in the n-sized heap a.

Output

Heap a with $n - 1$ elements.

Complexity

$O(\log n)$.

procedure HEAPDELETE(K)

$a[k] := a[n]$

$n := n - 1$

DownHeap(k)

6.6 Finding minimum key

Since minimum key is in the root of the heap, reading minimum is taking first element, putting the last one on it's place and heapifying the whole heap. Complexity is actually the complexity of DOWNHEAP.

Algorithm 27 Finding minimum in the binary heap

Input

n-sized heap a.

Output

Root (the largest) element of heap a, heap size decreased by one.

Complexity

$O(\log n)$.

procedure FINDMIN()

$k := a[1]$

$a[1] := a[n]$

$n := n - 1$

DownHeap(1)

return k

7 Leftist Heap

Leftist heap is a heap such that the left subtree has height not less than the right subtree's height. The motivation is to have a structure similar to the binary heap which also enables fast merging. It can be used for the union of priority queues.

The operations of interest are: inserting an element into heap, deleting it, getting minimum/maximum and merging two leftist heaps into one. The heap is represented as binary tree with keys in its nodes.

Null path length of a node v is length of the shortest path from v to a leaf. The length is defined recursively as

$$d(v) = \begin{cases} -1, v \text{ is null} \\ 1 + \min\{d(c_l), d(c_r)\}, \text{ otherwise} \end{cases}$$

where c_l and c_r are left and right children of v. *Leftist heap* H is the binary heap with property that $d(c_l) \geq d(c_r)$, for each $v \in H$.

In the figure 5, node 3 has null path length 1, node 8 has length 0, node 6 has length 2, and so on.

7.1 Merging heaps

Merging goes by nodes on the rightmost path and then fixing null path lengths.

Figure 3: Leftist heaps H_1 and H_2 to merge

Merging rightmost path starts from the root. At each step, H_1 is the heap with the lesser key at the root r_1. Then, the root r_2 becomes right child of r_1, and merging continues with $r_1.c_r$ and r_2. When $r_1.c_r$ and r_2 are merged, longer of them goes to the left and null path length is updated.

7.2 Other operations

Minimum key is in the root node, so finding minimum is getting the root key.
Deleting minimum key is removing root and merging its children.
Inserting key K into a heap H is realized as merging of H and single node heap with the key K.
Deleting a node x (not a key) goes by dropping x and merging its children.

7.3 Worst case complexity

Lemma 1. Leftist heap with $k \geq 1$ nodes on its rightmost path has at least $2^k - 1$ nodes.

Proof. For $k = 1$ theorem is trivially true. Suppose it holds for some k and let's proove it also holds for heap H with $k + 1$ elements on its rightmost path. If root is removed, then two leftist subheaps of k elements remain for which the inductive hypothesis holds true. Thus, for two subheaps there are $2(2^k - 1) = 2^{k+1} - 2$ nodes, so together with root gives $2^{k+1} - 1$ nodes in total in H. QED

Theorem 6. Complexity of merging algorithm for two heaps of total size of n elements is $O(\log n)$.

Algorithm 28 Merging leftist heaps

Input
H_1 First heap to merge.
H_2 Second heap to merge.

Output
H_1 Merged heaps in the first heap.

Complexity
$O(\log n)$ where n is total number of elements in H_1 and H_2.

procedure MERGE(H_1, H_2)
  if $H_1 =$ **null then**
    H_2
  if $H_2 =$ **null then**
    H_1
  if $r_1.k > r_2.k$ **then**
    Swap(r_1, r_2)
  if $r_1.c_r =$ **null then**
    $r_1.c_r = r_2$
  else
    $r_1.c_r :=$ Merge($r_1.c_r, r_2$)
    if $r_1.c_l.length < r_1.c_r.length$ **then**
      Swap($r_1.c_l, r_1.c_r$) ▷ Swap handles parent pointers.
    $r_1.length := r_1.c_r.length + 1$
  return H_1

Algorithm 29 Finding minimum in a leftist heap

Input
H n-sized heap.

Output
Minimum heap of H.

Complexity
$O(1)$.

procedure FINDMIN(H)
  return r_H

Algorithm 30 Deleting minimum in a leftist heap

Input
H n-sized heap.

Output
H without the minimum key.

Complexity
$O(\log n)$.

procedure DELETEMIN(H)
  if $r_H :=$ **null then**
    return
  $x := r_H$
  Merge($r_H.c_l, r_H.c_r$)
  return $x.k$

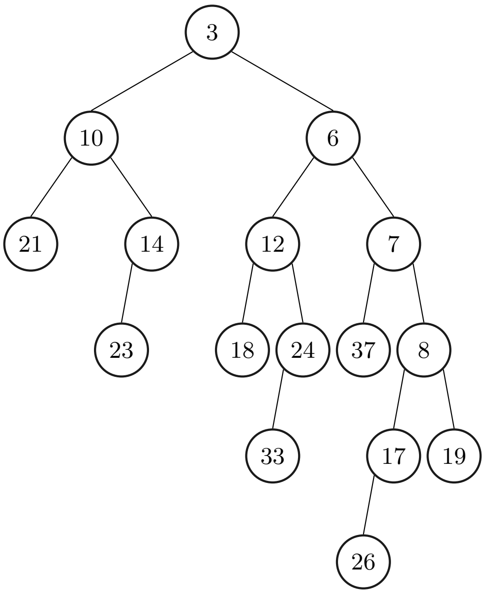

Figure 4: Merging over rightmost paths, fixing null path lengths at nodes 3 and 7

Algorithm 31 Inserting a key into a leftist heap

Input
- H Heap of the size n.
- K Key to be added into H.

Output
- H with newly added key K.

Complexity
- $O(\log n)$.

procedure INSERT(K)
- **new** x
- $x.c_l :=$ $x.c_r :=$ **null**
- $x.k :=$ **null**
- Merge(r_H, x)

Proof. According to the theorem 1, size of the rightmost path of n-sized leftist heap is $O(\log n)$. Since merging traverses nodes on the rightmost path, the proof follows trivially. QED

Complexity of the insert and delete are equal to the complexity of the merge operation.

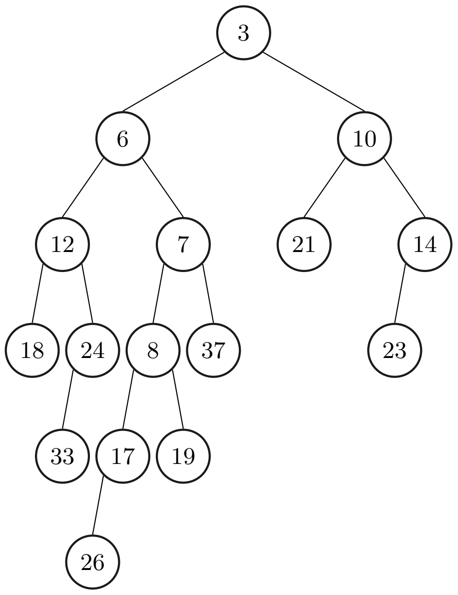

Figure 5: Fixed null path lengths

Algorithm 32 Deleting a node in a leftist heap

Input

H n-sized heap.

x Non-null node to delete.

Output

H without x.

Complexity

$O(\log n)$.

procedure DELETE(x)

Merge($x.c_l, x.c_r$)

8 Graph

Definitions for both directed and undirected graphs refer to graphs. Otherwise, a definition refers to directed or undirected graph.

A *directed graph* G is a pair (V, E), where V is finite set and E is a binary relation on V. The set V is called the *vertex set* of G, its elements are called *vertices*; the set $E = \{(x, y) : x, y \in V\}$ is called *edge set* of G, and its elements are called *edges*. Self-loops are allowed, i.e. it is possible to have edge $(y, y), y \in V$.

If the edge set E contains unordered pairs of distinct vertices, then $G = (V, E)$ is called *undirected graph*. That means that

$$E = \{\{x, y\} : x, y \in V, x \neq y\}$$

i.e. (x, y) and (y, x) are the same edge and self-loops are not allowed in an undirected graph.

For edge $(x, y) \in E$ of a directed graph $G = (V, E)$, one says that (x, y) is *incident from* (or *leaves*) vertex x and *incident to* (or *enters*) vertex y.

For edge $(x, y) \in E$ of an undirected graph $G = (V, E)$, one says that (x, y) is *incindent on* vertices x, y.

For edge $(x, y) \in E$ of a graph $G = (V, E)$, one says that y is *adjacent* to x. For an undirected graph, the adjacency relation is symmetric, which possibly may not be a case with directed graphs. Adjacent vertex y of x can be also written as $x \to y$.

In an undirected graph, *degree* of a vertex is number of edges incident on it.

In a directed graph, *out-degree* of a vertex is number of edges leaving it, *in-degree* is the number of edges entering it.

Sequence $(x_0, x_1, \ldots, x_k), k \geq 1$, of vertices $x_0, \ldots, x_k \in V$ such that $(x_{i-1}, x_i) \in E, i = 1, \ldots, k$, is called *path*. Number k is *path length*, which is equal to the number of edges. Path is *simple* if all vertices in the path are distinct. Subsequence of vertices $(x_i, x_{i+1}, \ldots, x_j), 0 \leq i \leq j \leq k$, is called *subpath* of the path $(x_0, x_1, \ldots, x_k)$. If for $x, y \in V$ there exists path p from x to y, then y is *reachable* from x via p.

In a directed graph, path $(x_0, x_1, \ldots, x_k), k > 0$, is a *cycle* if $x_0 = x_k$. If all vertices $x_1, \ldots, x_k$ are distinct then the cycle is *simple*. A self-loop is a cycle of length 1. Directed graph with no self-loops is *simple*.

In an undirected graph, path $(x_0, x_1, \ldots, x_k)$ is *simple cycle* if $k \geq 3$, $x_0 = x_k$ and $x_1, \ldots, x_k$ are distinct.

Graph with no cycles is *acyclic*.

Undirected graph is *connected* if for each pair of vertices there exists a path which connects them. The relation "is reachable" is the relation of equivalence. Therefore, it splits non-connected undirected graph into classes of equivalence, which are called *connected components*. In other words, undirected graph is connected if and only if it contains only one connected component.

In a directed graph $G = (V, E)$, vertices $x, y \in V$ are *mutually reachable* if there exist paths from both x to y and from y to x. G is *strongly connected* if each two vertices are mutually reachable. The *mutually reachable* is relation of equivalence, so it splits non strongly connected graph into classes of equivalence, which are called *strongly connected components*. In other words, directed graph is strongly connected if and only if it contains only one strongly connected component.

A digraph G is *weakly connected* if the undirected *underlying graph* obtained by replacing all directed edges of G with undirected edges is a connected graph. A digraph G is *connected* if for each two $x, y \in V$ there exists $x \rightsquigarrow y$ or $y \rightsquigarrow x$. It is trivial to prove that these definitions are equivalent.

A directed graph $G = (V, E)$ is *singly connected* if $x \rightsquigarrow y$ implies that there is at most one simple path from x to y for all vertices $x, y \in V$.

The *transpose* of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T), E^T = \{(y, x) \in V \times V : (x, y) \in E\}$. G^T is computed from G in $\Theta(V + E)$ time, if G is represented with adjacency list.

The *square* of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$, where $E^2 = \{(x, z) \in V \times V : (\exists y \in V)(x, y) \in E, (y, z) \in E\}$. G^2 is computed from G in $\Theta(V + E)^2$ time, if G is represented with adjacency list.

A *cut* is a partition of vertices of a graph into two disjoint subsets. The *cut-set* of the cut is the set of edges whose end points are in different subsets of the partition. An edge *crosses* the cut if they are in its cut-set. A cut *respects* an edge set A if no edge from A crosses the cut.

8.1 Tree

A *forest* is undirected, acyclic graph.

Tree is undirected, connected, acyclic graph. Its vertices are also called *nodes*. It is obvious that for each two nodes in the tree, there exists only one path which connects them.

Rooted tree is a tree where one vertex is distinguished from the others. That vertex is called *root* of the tree. Thus, for each node x in the tree T with root r, there exists unique path from r to x. Any node $y \neq$ on that path is *ancestor* of x, i.e. x is *descendant* of y. If (y, x) is the last edge on that path, then y is parent of x, and x is child of y. Root has no parent. Nodes with same parent are *siblings*. Node with no children is *leaf* (or *external node*). Non-leaf node is *internal node*.

Subtree rooted at node x is tree with root x and all its descendants.

Ordered tree is a tree where an order is imposed, i.e. if node has k children, then one can distinguish first child, second child, $\ldots$, k-th child.

Number of children of the node x of the tree T is called *degree* of node x. Length of a path from root r to x is x's *depth* in T. *Height of node x* in the tree T is the number of edges on the longest simple path from x to any leaf. *Height of tree* is height of the root r.

Binary tree is a tree where each node has at most two children.

Full binary tree (also called *strict binary tree*) is a tree in which every node other than the leaf has two children.

Complete binary tree is a binary tree in which every level (except possibly the last one) is completely filled, and all nodes are as far left as possible.

Max heap is a tree that satisfies the *max heap property*: if b is a child node of a, then key $a.k \geq b.k$. *Min heap* is a tree that satisfies the *min heap property*: if b is a child node of a, then key $a.k \leq b.k$.

Binary heap is a complete binary tree with a heap property.

Operations of interests for particular data structure D are: inserting, deleting and updating value in D, searching for given value v in D, finding minimum and maximum values in D, union of two data structures D_1 and D_2 into new data structure of the same type.

8.2 Breadth first search

The algorithms starts from a given vertex s, which is taken as the source to start the BFS algorithm. The search goes by visiting all neighbors from s, in case they are not visited yet. When these neighbors are visited, then neighbors of each of them is visited in the same way. For the purpose of tracing the neighbors, a queue Q can be used. To determine whether a vertex is being visited or not, an attribute $x.v$ can be used with the values: N - still not visited, Y - visit finished, P - the visit is in progress. Each vertex has the previous vertex computed in $x.p$. The distance of x to s is stored in $x.d$.

8.3 Depth first search

The algorithms starts from a given vertex s, which is taken as the source to start the DFS algorithm. The search goes by taking the first non-visited neighbor and start DFS recursively on it. When no such neighbor is available anymore, the recursion stops. For each $x \in V$, the algorithm traces discovering and finishing time on x in the attributes $x.t_d$ and $x.t_f$.

8.4 Topological sort

On a directed acyclic graph, it is possible to impose linear ordering on vertices, such that if $(x, y) \in E$, then x appears before y in that ordering.

Algorithm 33 Breadth first search

Input

G Graph to search. For all $x \in V$ it's set by default $v = \mathrm{N}, d = \infty, p = \mathrm{null}$. Its adjacent vertices are in the set A.

Input

s Source vertex to start the search.

Output

G For each $x \in V$ the attributes v, d, p are computed.

Complexity

$O(|E| + |V|)$

```
procedure BreadthFirstSearch(s)
    s.v = P, s.d = 0, s.p = null
    Q = ∅
    Q.push(s)
    while Q ≠ ∅ do
        x = Q.pop()
        for y := x.A do
            if y.v = N then
                y.v := P
                y.d := x.d + 1
                y.p = x
                Q.push(y)
        x.v = Y
```

Algorithm 34 Depth first search

Input

G Graph to search. For all $x \in V$ it's set by default $x = \mathrm{N}, t_d = \infty, t_f = \infty$, $p = \mathrm{null}$. Its adjacent vertices are in the set A.

Input

s Source vertex to start the search.

Output

G For each $x \in V$ the attributes v, d, p, t_d, t_f are computed.

Complexity

$O(|E| + |V|)$

```
procedure DepthFirstSearch(s)
    t := 0
    for x ∈ V do
        if x.v = no then
            DfsVisit(x)
procedure DfsVisit(x)
    t := t + 1
    x.t_d = t, x.v = P
    for y ∈ x.A do
        if y.v = N then
            y.p = x
            DfsVisit(y)
    t := t + 1
    u.v = Y, u.t_f = t
```

Algorithm 35 Topological sort

Input

G Graph to sort.

Output

List of vertices topologically sorted.

Complexity

$O(|E| + |V|)$

procedure TopologicalSort(s)
  $L = \emptyset$ ▷ List of vertices.
  Call DepthFirstSearch(G) to compute $x.t_f$ for $x \in V$.
  As each x is finished, insert it at the front of L.
  return L

8.5 Strongly connected components

Graphs G and G^T have exactly the same strongly connected components: $x \leadsto y$ iff $y \leadsto x$.

Algorithm 36 Strongly connected components

Input

G Graph to compute SCCs.

Output

Strongly connected components.

Complexity

$O(|E| + |V|)$

procedure StronglyConnectedComponents(s)
  $L := \emptyset$ ▷ List of components.
  DepthFirstSearch(G)
  DepthFirstSearch(G^T) but in the main loop of DFS take $x \in V$ in order of decreasing $x.t_f$.
  Each tree of the DFS forest of G^T put into L.
  return L

9 AVL tree

Binary search tree is the *AVL tree* if for each node, difference between the left and right subtree height is less or equal than 1. For n nodes, an AVL tree has height of $1.44 \lg n$. For that reason, finding, inserting or deleting node is of $O(\log n)$ complexity. The presented algorithms assume that all keys stored in the tree are different.

Lemma 2. Maximum height of AVL tree with n nodes is $1.44 \lg n$.

Proof. To find maximum height of an AVL tree with n nodes, one should answer what is the minimum number of nodes (sparsest possible AVL tree) an AVL tree of height h can have? Let F_h be an AVL tree of height h, having the minimum number of nodes. Let F_l and F_r be AVL trees which are left and right subtree, respectively, of F_h. Then F_l or F_r must have height $h-2$. Suppose F_l has height $h-1$ so that F_r has height $h-2$. F_l has to be an AVL tree having the minimum number of nodes among all AVL trees with height of $h-1$ and F_r among all AVL trees of height $h-2$. Thus, $|F_h| = |F_{h-1}| + |F_{h-2}| + 1$, where $|F_h|$ denotes number of nodes in F_h. Such trees are called Fibonacci trees. Note that $|F_0| = 1$ and $|F_1| = 2$ and $|F_h| + 1 = (|F_{h-1}| + 1) + (|F_{h-2}| + 1)$, so $|F_h|$ are Fibonacci numbers. Using the approximate formula for Fibonacci numbers, we get

$$|F_h| + 1 \approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+3} \Rightarrow h \approx 1.44 \lg |F_n|$$

This implies that the sparsest possible AVL tree with n nodes has height $h \approx 1.44 \lg n$ which is the worst case of AVL tree's height. QED

9.1 Finding node

Finding a node with a given key K starts from the root r_T as current node. K is compared with a key of each current node. If it is less than it's value, it continues within left subtree; if it's greater, then proceeds within right subtree.

Algorithm 37 Finding a key in an AVL tree.

Input
  K Key to find.
Output
  Node with the key K or null if no K is present or T is empty.
Complexity
  $O(\lg n)$

```
procedure FIND(K)
    x = r_T
    while x ≠ null do
        if K < x.k then
            x = x.c_l
        else if K > x.k then
            x = x.c_r
        else
            return x
    return null
```

To find a predecessor of a given node x, one should get the most right descendant of x's left child $x.c_l$. From this definition, it follows that predecessor has no right child (otherwise, that child would be the predecessor).

To find a successor of a given node x, one should get the most left descendant of x's right child $x.c_r$. From this definition, it follows that successor has no left child (otherwise, that child would be the successor).

Algorithm 38 Finding a predecessor of a given node.

Input

x Node to find predecessor of.

Output

Predecessor node or null (if x is null or has no predecessor or T is empty).

Complexity

$O(\lg n)$

procedure PREDECESSOR(x)
  if $x =$ **null or** $r_T =$ **null or** $x.c_l =$ **null then**
    return null
  $x_l := x.c_l$
  while $x_l \neq$ **null do**
    $x_l := x_l.c_r$
  return x_l

Algorithm 39 Finding a successor of a given node.

Input

x Node to find succecessor of.

Output

Successor node or null (if x is null or has no successor or T is empty).

Complexity

$O(\lg n)$

procedure SUCCESSOR(x)
  if $x =$ **null or** $r_T =$ **null or** $x.c_r =$ **null then**
    return null
  $x_r := x.c_r$
  while $x_r \neq$ **null do**
    $x_r := x_r.c_l$
  return x_r

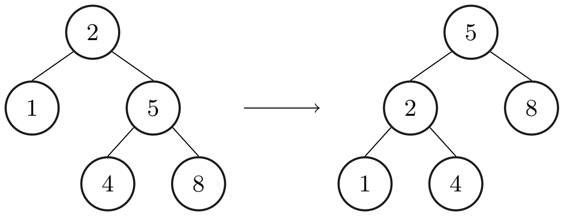

Figure 6: Left rotation of node 2

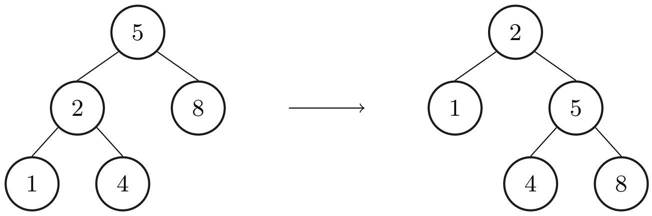

Figure 7: Right rotation of node 5

One can easily check if a node is descendant of an another node.

Algorithm 40 Checking whether a node is descendant of another node.

Input
- a Ancestor node.
- d Descendant node.

Output
- True if d is descendant of a.

Complexity
- $O(\lg n)$

procedure IsLeftDescendant(a, d)
    while $d.p \neq a$ **do**
        $d := d.p$
    if $d = a.c_l$ **then**
        return true
    else
        return false

9.2 Rotations

Rotations reconnect nodes as described below in figures 6 and 7. There are no changes on balance factors, they are fixed in the appropriate operations.

Left rotation reconnects nodes, such that rotated node x get it's right child $x.c_r$ for the parent and left child of $x.c_r$ becomes right child of x. Node 2 in the following figure is rotated to the left:

Right rotation reconnects nodes, such that rotated node x get it's left child for the parent and right child of $x.c_l$ becomes left child of x. Node 5 in the following figure is rotated to the right:

9.3 Inserting node

Inserting node is to put the new key K into tree T by going to the left subtree if K is less than key of the current node, and to the right if K is greater than key of the current node. When a node is inserted, balance factors of some of the traversed nodes can be changed. For that reason, those nodes have to be rebalanced.

While searching correct place to insert new node, last node P with non-zero balance is memorized. If such node does not exist, then no balancing is necessary after inserting the new node. Subtree at

Algorithm 41 Left rotation of a node.

Input

x Node to rotate.

Output

T Tree with the left rotated x.

Complexity

$O(1)$

procedure RotateLeft(x)

$x_p := x.p$
$x_r := x.c_r$
$x_{rl} := x.c_r.c_l$
$x_r.p := x.p$
$x.p := x_r$
if $x_{rl} \neq$ **null then**
  $x_{rl}.p := x$
$x.c_r := x_{rl}$
$x_r.c_l := x$
if $x_p \neq$ **null then**
  if $x = x_p.c_l$ **then**
    $x_p.c_l := x_r$
  else if $x = x_p.c_r$ **then**
    $x_p.c_r := x_r$
if $r_T = x$ **then**
  $r_T := x.p$

Algorithm 42 Right rotation of a node.

Input

x Node to rotate.

Output

T Tree with the right rotated x.

Complexity

$O(1)$

procedure RotateRight(x)

$x_p := x.p$
$x_l := x.c_l$
$x_{lr} := x.c_l.c_r$
$x_l.p := x.p$
$x.p := x_l$
if $x_{lr} \neq$ **null then**
  $x_{lr}.p := x$
$x.c_l := x_{lr}$
$x_l.c_r := x$
if $x_p \neq$ **null then**
  if $x = x_p.c_l$ **then**
    $x_p.c_l := x_l$
  else if $x = x_p.c_l$ **then**
    $x_p.c_l := x_l$
if $r_T = x$ **then**
  $r_T := x.p$

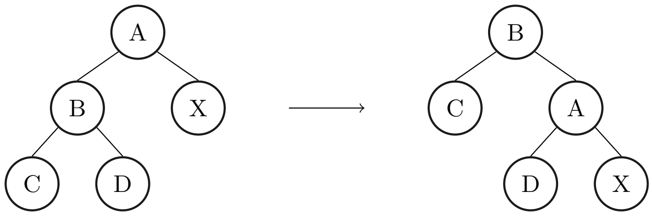

Figure 8: L1 case fixed with right rotation of A

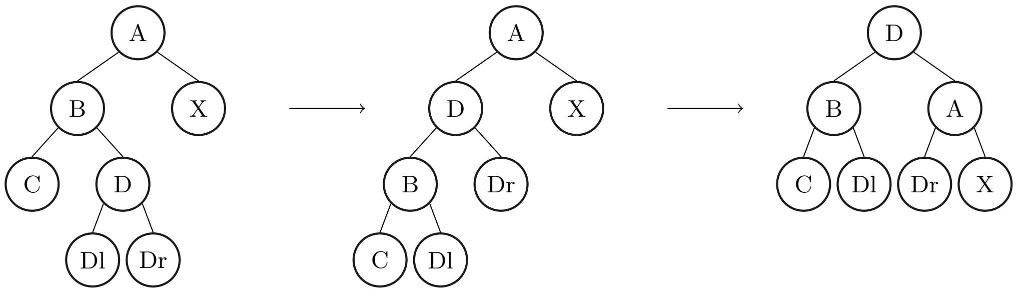

Figure 9: L2, L3, L4 cases fixed with left rotation of B and right rotation of A

P can become corrupted after inserting new node and no other tree except this one can be corrupted. Balances of all nodes from the new one until P are modified. Rotations are made in constant time, so total time for inserting is $O(\lg n)$.

Let A be the last node with the non-zero balance $A.b \neq 0$, let the new node be inserted into left subtree of A, let be $B = A.c_l, C = B.c_l, D = B.c_r, X = A.c_r$ and $x.h$ be the height of a subtree at node x.

First case is when the inserted node is left descendant of B. Then, $A.b = -2, B.b = -1$, so $D.h = X.h, C.h = D.h + 1 \Rightarrow C.h = X.h + 1$. If A is right rotated, then B is parent of A, D and X are children of A. It follows that $A.b = 0$ since $D.h = X.h$ and $B.b = 0$ because $C.h = X.h + 1$.

Second, third and fourth case are when inserted node is right descendant of B. Three possibilities are available: $A.b = -2, B.b = +1, D.b = -1$, $A.b = -2, B.b = +1, D.b = +1$ or $A.b = -2, B.b = +1, D.b = 0$, depending of where the node inserted (left or right subtree of D). Denote $D_l = D.c_l, D_r = D.c_r$, and consider the second case $D.b = -1$. It follows that $b(B) = 1 \Rightarrow D.h = C.h + 1, D.b = -1 \Rightarrow D_l.h = D_r.h + 1, h(D) = h(D_l) + 1$, so $D_l.h = C.h$. Also, $X.h = B.h - 2 = D.h - 1 = D_l.h = D_r.h + 1$. Therefore, $D_l.h = C.h \Rightarrow B.b = 0; X.h = D_r.h + 1 \Rightarrow A.b = +1; D_l.h = X.h \Rightarrow D.b = 0$.

For the third case, rotations are the same and calculus is similar: $C.h = D_r.h = D_l.h + 1, X.h = D_r.h \Rightarrow B.b = -1, A.b = 0, D.b = 0$. The fourth case is same as this one. Let's write down these cases symbolically:

Symmetric cases come when inserted node is in the right subtree of A.

The following functions perform cases L1-L3, R1-R3.

Algorithm 43 Case L1.

Input

x Subtree to perform the case L1.

Output

None.

Complexity

$O(1)$

procedure CaseL1(x)
$A := x, B := A.c_l$
RotateRight(A)
$A.b := 0, B.b := 0$

Algorithm 44 Case L2.

Input

x Subtree to perform the case L2.

Output

None.

Complexity

$O(1)$

procedure CASEL2(x)

$A := x, B := A.c_l, D := B.c_r$

RotateLeft(B)

RotateRight(A)

$A.b := +1, B.b := 0, D.b := 0$

Algorithm 45 Case L3.

Input

x Subtree to perform the case L3.

Output

None.

Complexity

$O(1)$

procedure CASEL3(x)

$A := x, B := A.c_l, D := B.c_r$

RotateLeft(B)

RotateRight(A)

$A.b := 0, B.b := -1, D.b := 0$

Algorithm 46 Case L4.

Input

x Subtree to perform the case L4.

Output

None.

Complexity

$O(1)$

procedure CASEL4(x)

$A := x, B := A.c_l, D := B.c_r$

RotateLeft(B)

RotateRight(A)

$A.b := 0, B.b := 0, D.b := 0$

Algorithm 47 Case R1.

Input

x Subtree to perform the case R1.

Output

None.

Complexity

$O(1)$

procedure CASER1(x)

$A := x, B := A.c_r$

RotateLeft(A)

$A.b := 0, B.b := 0$

Algorithm 48 Case R2.

Input

x Subtree to perform the case R2.

Output

None.

Complexity

$O(1)$

procedure CaseR2(x)
  $A := x, B := A.c_r, D := B.c_l$
  RotateRight(B)
  RotateLeft(A)
  $A.b := -1, B.b := 0, D.b := 0$

Algorithm 49 Case R3.

Input

x Subtree to perform the case R3.

Output

None.

Complexity

$O(1)$

procedure CaseR3(x)
  $A := x, B := A.c_r, D := B.c_l$
  RotateRight(B)
  RotateLeft(A)
  $A.b := 0, B.b := +1, D.b := 0$

Algorithm 50 Case R4.

Input

x Subtree to perform the case R4.

Output

None.

Complexity

$O(1)$

procedure CaseR4(x)
  $A := x, B := A.c_r, D := B.c_l$
  RotateRight(B)
  RotateLeft(A)
  $A.b := 0, B.b := 0, D.b := 0$

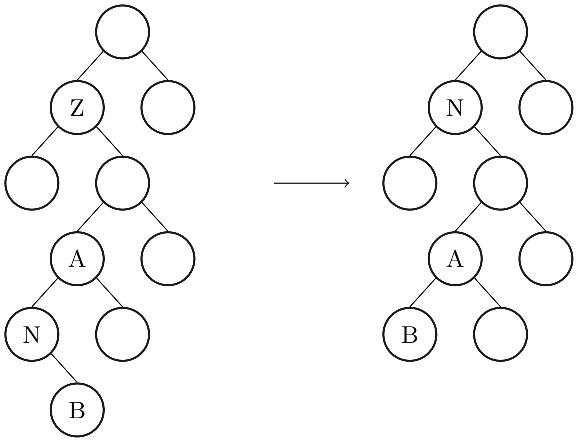

Figure 10: Deleting node Z by replacing it with successor N.

Inserting searches for an appropriate leaf node to put the key K into one of it's children. Then the balance factors of the traversed nodes are fixed.

9.4 Deleting node

When a node is deleted, heights of subtrees containing that node may be changed. For that reason, rebalancing has to be performed of all nodes from a deleted one until the root. Deleting Z if both children are null is removing it and checking all parents for balances. If some of the Z's children isn't null, then deleting it is replacing it with predecessor or successor node (call it N). N's parent A takes N's single child B as a new child instead of N, Z is replaced with N. The procedure is shown on the figure 10.

Nodes starting from A should be checked for balances and rotated if necessary. If A's height has not changed (balance is 0), the deleting procedure ends; otherwise, A becomes A's parent and procedure is repeated. There are three cases on deleting:

D1 $A.b = 0$, after deleting $A.b = \pm 1$ and height of A-tree is not changed, so the deleting procedure is ended.

D2 $A.b = \pm 1$, after deleting $A.b = 0$, so there's no need for rotations; but height of trees containing A is changed, so procedure of balancing continues on parent of A.

D3 $A.b = \pm 1$, after deleting $A.b = \pm 2$, so rotation are made; height of trees containing A is changed, so procedure of balancing continues on parent of A. L1-L5 and R1-R5 cases are possible here.

Additional cases on deleting are:

L5 $A.b = -2, B.b = 0 \Rightarrow A.c_r \Rightarrow A.b = -1, B.b = +1$

R5 $A.b = +2, B.b = 0 \Rightarrow A.c_l \Rightarrow A.b = +1, B.b = -1$

Rotations are made in $O(\lg n)$ time, so as finding node to delete, so total time for deleting is $O(\lg n)$.

Algorithm 51 Inserting a node with the given key.

Input
K Key to insert.

Output
T Tree with the newly added key.

Complexity
$O(\lg n)$

procedure INSERT(K)
    new z
    $z.k := K$
    ▷ Empty tree is trivial case.
    if $r_T =$ **null then**
        $r_T := z$
        return
    ▷ Not empty tree.
    $c := r_T$
    $P :=$ **null** ▷ Last ancestor with non-zero balance.
    ▷ Insert z into an empty place.
    while true do
        if $c.b \neq 0$ **then**
            $P := c$
        if $K < c.k$ **then**
            if $c.l =$ **null then**
                $c.l := z$
                $z.p := c$
                break
            $c := c.c_l$
        else
            if $c.c_r =$ **null then**
                $c.c_r := z$
                $z.p := c$
                break
            $c := c.c_r$
    if $P =$ **null or** $P.b = 0$ **then** ▷ Just modify balances.
        $c := z$
        do
            if $c = c.p.l$ **then**
                $c.p.b := c.p.b - 1$
            else
                $c.p.b := c.p.b + 1$
            $c := c.p$
        while $c \neq r_T$
        return
    ▷ Modify balances from z to P.
    $c := z$
    do
        if $c = c.p.l$ **then**
            $c.p.b := c.p.b - 1$
        else
            $c.p.b := c.p.b + 1$
        $c := c.p$
    while $c \neq P$

Algorithm 52 Inserting a node with the given key.

▷ Fix balance factors.
if IsLeftDescendant(P, c) = **true then** ▷ Node inserted to the left.
  $A := P, B := P.c_l, D := B.c_r$
  if $A.b = -2$ **and** $B.b = -1$ **then**
    CaseL1(A)
  else if $A.b = -2$ **and** $B.b = +1$ **and** $D.b = -1$ **then**
    CaseL2(A)
  else if $A.b = -2$ **and** $B.b = +1$ **and** $D.b = +1$ **then**
    CaseL3(A)
  else if $A.b = -2$ **and** $B.b = +1$ **and** $D.b = 0$ **then**
    CaseL4(A)
else▷ Node inserted to the right.
  $A := P, B := P.c_r, D := B.c_l$
  if $A.b = +2$ **and** $B.b = +1$ **then**
    CaseR1(A)
  else if $A.b = +2$ **and** $B.b = -1$ **and** $D.b = +1$ **then**
    CaseR2(A)
  else if $A.b = +2$ **and** $B.b = -1$ **and** $D.b = -1$ **then**
    CaseR3(A)
  else if $A.b = +2$ **and** $B.b = -1$ **and** $D.b = 0$ **then**
    CaseR4(A)

Algorithm 53 Case L5.

Input
  x Subtree to perform the case L5.
Output
  None.
Complexity
  $O(1)$
procedure CASEL5(x)
  $A = x, B = A.c_l$
  $RotateRight(A)$
  $A.b = -1, B.b = +1$

Algorithm 54 Case R5.

Input
  x Subtree to perform the case R5.
Output
  None.
Complexity
  $O(1)$
procedure CASER5(x)
  $A = x, B = A.c_r$
  $RotateLeft(A)$
  $A.b = +1, B.b = -1$

Algorithm 55 Deleting a node for the given key.

Input
  K Key to delete.
Output
  None.
Complexity
  $O(\lg n)$

procedure CASER4(x)
  if $r_T =$ **null then**
    return
  $Z :=$ find(K)
  if $Z =$ **null then**
    return
  if $Z = r_T$ **then**
    delete r_T
  $N_s =$ Successor(Z), $N_p =$ Predecessor(Z)
  $A =$ **null** ▷ Parent of N_s or N_p.
  if $N_s =$ **null and** $N_p =$ **null then**
    $A = p(Z)$
    if $Z = A.c_l$ **then**
      $A.b := A.b + 1$
      $A.c_l := Null$
    else if $Z = A.c_r$ **then**
      $A.b := A.b - 1$
      $A.c_r := Null$
    delete Z
  else if $N_s \neq$ **null then**
    $Z.k := N_s.k$
    $A := N_s.p$ ▷ Could be also $A = Z.p$.
    if $N_s = A.c_l$ **then** ▷ Successor is not sibling of Z.
      if $N_s.c_r \neq$ **null then** ▷ Connect A with single child (if exists).
        $A.c_l := N_s.c_r$
        $N_s.c_r.p := A$
      else
        delete $A.c_l$
      $A.b := A.b + 1$
    else if $N_s = A.c_r$ **then** ▷ Successor is sibling of Z.
      if $N_s.c_r \neq$ **null then**
        $A.c_r := N_s.c_r$
        $N_s.c_r.p := A$
      else
        delete $A.c_r$
      $A.b := A.b - 1$
    delete N_s

Algorithm 56 Deleting a node for the given key.

```
    else if $N_p \neq$ null then
        $Z.k := N_p.k$
        $A := N_p.p$ ▷ Could be also $A = Z.p$.
        if $N_p = A.c_r$ then ▷ Successor is not sibling of $Z$.
            if $N_p.c_l \neq$ null then ▷ Connect $A$ with single child (if exists).
                $A.c_r := N_p.c_l$
                $N_p.c_l.p = A$
            else
                delete $A.c_r$
            $A.b := A.b - 1$
        else if $N_p = A.c_l$ then ▷ Successor is sibling of $Z$.
            if $N_p.c_l \neq$ null then
                $A.c_l := N_p.c_l$
                $N_p.c_l.p := A$
            else
                delete $A.c_l$
            $A.b := A.b + 1$
        delete $N_p$
    ▷ Correct balances along the tree starting from parent.
    while $A \neq$ null do
        if $A.b = \pm 1$ then ▷ Case D1.
            break
        else if $A.b = 0$ then ▷ Case D2.
            if $A.p \neq$ null then
                if $A = A.p.l$ then
                    $A.p.b := A.p.b + 1$
                else if $A = A.p.c_r$ then
                    $A.p.b := A.p.b - 1$
        else if $A.b = \pm + 2$ then ▷ Cases R1 - R5.
            $B := A.c_r$
            if $B.b = +1$ then
                CaseR1($A$)
            else if $B.b = -1$ then
                $D := B.c_l$
                if $D.b = +1$ then
                    CaseR2($A$)
                else if $D.b = -1$ then
                    CaseR3($A$)
                else if $D.b = 0$ then
                    CaseR4($A$)
                else if $B.b = 0$ then
                    CaseR5($A$)
```

Algorithm 57 Deleting a node for the given key.

```
else if A.b = ±−2 then ▷ Cases L1 - L5.
    B := A.c_l
    if B.b = −1 then
        CaseL1(A)
    else if B.b = +1 then
        D := B.c_r
        if D.b = −1 then
            CaseL2(A)
        else if D.b = +1 then
            CaseL3(A)
        else if D.b = 0 then
            CaseL4(A)
        else if B.b = 0 then
            CaseL5(A)
A := A.p
```

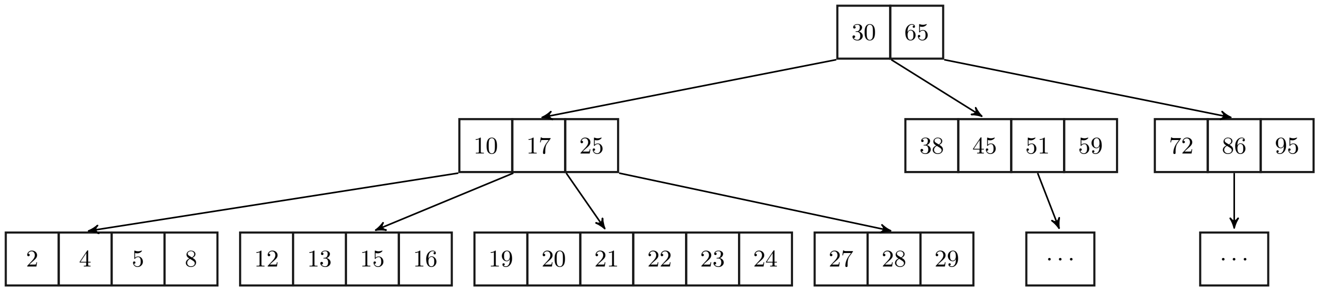

Figure 11: Example of B tree of degree $t = 4$

10 B tree

Motivation for B the tree is to have data structure that seldom reads or writes keys from the external memory. When B tree does the read or write, keys are taken in batches, so the communication with the external memory is minimized. Operations of interest are finding, inserting and deleting key.

Definition 10.1. *B tree T with a root r is a tree with the following properties:*

1. Every node x has the following fields:
   (a) n – number of keys currently stored in node x.
   (b) k_i – keys stored in nondecreasing order, so that $k_1 \leq k_2 \leq \ldots \leq k_n$.
   (c) l – boolean which is true if x is a leaf and false if x is an internal node.
2. Each internal node x contains $n + 1$ children $c_1, c_2, \ldots, c_{n+1}$. Leaf nodes have no children, so those fields are null.
3. The keys k_i separate the ranges of keys stored in each subtree; if m_i is any key stored in the subtree with root $c_i, 1 \leq i \leq n$, then

$$m_1 \leq k_1 \leq m_2 \leq k_2 \leq \ldots \leq k_n \leq m_{n+1}$$

4. All leaves have the same depth, which is the tree's height $T.h$.
5. Each internal node except the root contains at least $t - 1$ and at most $2t - 1$ keys. If tree is nonempty, then root has at least one key. Integer $t \geq 0$ is called node degree.
6. Every node x is read from an external memory by calling Read(x) and written by calling Write(x).

10.1 Searching

To find a key K in a subtree at node x, the given node is checked for existence of such key. If not found, the correct subtree c_i is determined to check recursively. Adjacent keys k_i and k_{i+1} such that $k_i \leq K \leq k_{i+1}$ are found, then searching is continued on c_i.

Finding index i at node x such that $K = k_i$ for the given key K is trivial.

Finding index i such that given key K fits into c_i's keys range is also trivial.

Finding predecessor key of the given k_i in node x is finding the right most key in the subtree c_i. Similarly, finding successor key of the given k_i in node x is finding the left most key in the subtree c_{i+1}.

10.2 Auxiliary node operations

Splitting child node c_i of x is an operation performed on a full node c_i ($n = 2t-1$ where n is number of keys in c_i) and x is not full. Splitting moves central key (the one at t-th place) to the correct place at the parent. The picture shows splitting node of seven keys to two nodes of three, while key **26** is moved up.

Algorithm 58 Finding a key

Input

K Key to find in a subtree x.

x Subtree to search.

Output

Node which contains K or null.

Complexity

$O(\log |T|)$

```
procedure FindKey(K, x)
    z := x
    while z ≠ null do
        i := 1
        while i ≤ z.n and K > z.k_i do
            i := i + 1
        if i ≤ z.n and K = z.k_i then
            break ▷ z is found.
        if z.l = true then
            z = null ▷ z is not found.
        else
            z := Read(z.c_i) ▷ Get child from an external memory.
        return z
```

Algorithm 59 Finding an index

Input

K Key to find in node x.

x Node x of degree t to search in.

Output

Index i of x or null.

Complexity

$O(t)$

```
procedure FindIndex(K, x)
    for i := 1 to x.n do
        if K = x.k_i then
            return i
    return null
```

Algorithm 60 Finding an index of a child

Input

K Key K to find a corresponding child.

x Node x of degree t to search in.

Output

Index i such that K belongs to $x.c_i$ or null.

Complexity

$O(t)$

```
procedure FindIndexChild(K, x)
    for i := 1 to x.n do
        if x.k_i ≤ K ≤ x.k_{i+1} then
            return i
    return null
```

Algorithm 61 Predecessor of a node

Input

x Node where to look for the predecessor.

i Index of the node x.

Output

Predecessor key determined as node y and index j, null if not found.

Complexity

$O(\log |T|)$

```
procedure PREDECESSOR(x, i)
    if x.l then
        y := x
        if i = 1 then
            (y, j) := null
        else
            j := i − 1
    else
        x = Read(x.c_i)
        while not x.l do
            x = Read(x.c_n)
        y := x, j := x.n
    return (y, j)
```

Algorithm 62 Successor of a node

Input

x Node where to look for the successor.

i Index of the node x.

Output

Successor key determined as node y and index j, null if not found.

Complexity

$O(\log |T|)$

```
procedure SUCCESSOR(x, i)
    if x.l then
        y := x
        if i = x.n then
            (y, j) := null
        else
            j := i + 1
    else
        x = Read(x.c_{i+1})
        while not x.l do
            x = Read(x.c_1)
        y := x, j := 1
    return (y, j)
```

Algorithm 63 Splitting a node

Input

x Node of the degree at least t.

i Index of the full child to split.

Output

Node x split at i-th child.

Complexity

$O(t)$

procedure Split(x, i)
  $y := x.c_i$ ▷ Full node.
  new z
  $z.l := y.l$
  $z.n := t - 1$
  ▷ Copy second half of keys from y to z.
  for $j := 1$ **to** $t - 1$ **do**
    $z.k_j := y.k_{t+j}$
  ▷ Copy second half of children from y to z.
  if not $y.l$ **then**
    for $j := 1$ **to** t **do**
      $z.c_j := y.c_{t+j}$
  $y.n := t - 1$
  ▷ Move x's children one place to the right to make room for z.
  for $j := x.n + 1$ **downto** $i + 1$ **do**
    $x.c_{j+1} := x.c_j$
  $x.c_{i+1} := z$
  ▷ Add new key $y.k_t$ for z into x.
  for $j := x.n$ **downto** i **do**
    $x.k_{j+1} := x.k_j$
  $x.k_i := y.k_t$
  $x.n := x.n + 1$
  Write(x)
  Write(y)
  Write(z)

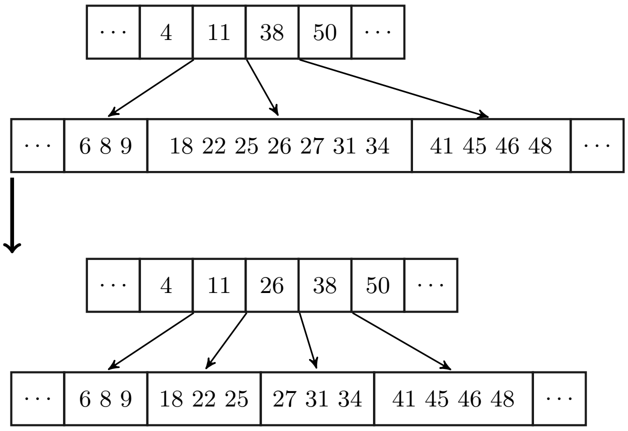

Figure 12: splitting 7-elements node ($t = 4$)

Merging is an operation reversed to the split operation. For a node x with at least t keys and children c_i and c_{i+1} with $t-1$ keys – the key k_i of x, all keys k_j of c_i and all keys k_l of c_{i+1} (where $1 \leq j, l \leq t-1$) are collapsed into single c_i node with $2t-1$ keys. The picture is analogous to the one of splitting node.

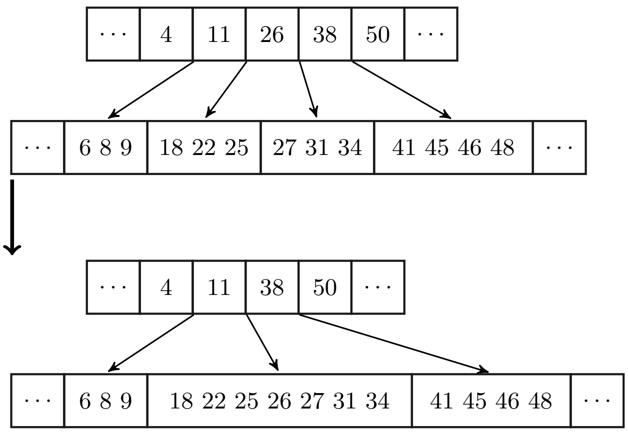

Figure 13: merging two 3-elements nodes ($t = 4$)

Key can be moved from node a (assuming that number of keys is not less than t) to immediate sibling b (assuming that number of keys is less than $2t-1$). Let x be their common parent, so $a = c_i$ and $b = c_{i+1}$ for some i; let p_j be the last key in a which is going to be moved. Since

$$K \leq p_j \leq k_i \leq L \leq k_{i+1} \text{ for all } K \in c_i, L \in c_{i+1}$$

p_j becomes the new k_i and old k_i becomes the first key q_1 in b. Old keys in b are moved one place to the right, as well b's children if b is not leaf. Also if a is not leaf, then it's child d_j can stay on it's own place but d_{j+1} has to be moved. Because new k_i has value of p_j and new q_1 has value of old k_i, without violating B tree properties it can be set $e_1 = d_{j+1}$ (e_1 is the first child in b). Since k_i is the only key affected by moving and $a = c_i$, $b = c_{i+1}$, no child of x is moved to the right. The picture shows moving of key **36**.

Symetrically, first key from node $a = c_i$, $2 \leq i \leq n+1$, with the number of keys not less that t, can be moved to immediate sibling $b = c_{i-1}$, with the number of keys less that $2t-1$.

10.3 Inserting

Inserting key into B tree is about finding appropriate non-full leaf node to insert the key. To insert key K into non-full node x, check if x is leaf – if does, find the right place to insert; if not, then insert into a child where K belongs.

Algorithm 64 Merging a node

Input

x Node to merge.

i Index of x to merge $x.c_i$ and $x.c_{i+1}$.

Output

Merged children $x.c_i$ and $x.c_{i+1}$.

Complexity

$O(t)$

procedure MERGE(x, i)
  $y := x.c_i, z = x.c_{i+1}$
  ▷ Move i-th key of x into y.
  $y.k_t := x.k_i$
  ▷ Move the rest of x's keys to the left.
  for $j := i$ **to** $x.n$ **do**
    $x.k_j := x.k_{j+1}$
  delete $x.k_{n+1}$
  $x.n := x.n - 1$
  ▷ Copy z's keys into y.
  for $j := 1$ **to** $t - 1$ **do**
    $y.k_{t+1} := z.k_j$
  ▷ Copy z's children into y.
  if not $z.l$ **then**
    for $j := 1$ **to** t **do**
      $y.c_{t+j} := z.c_j$
  $y.n = 2 \cdot t - 1$
  delete z
  ▷ Remove link for z from x.
  delete $x.c_{i+1}$
  for $j := i + 1$ **to** $x.n$ **do**
    $x.c_j := x.c_{j+1}$
  delete $x.c_n$
  Write(x)
  Write(y)
  Write(z)

Algorithm 65 Moving key to a next child

Input

x Node of the degree t where the key is moved between the adjacent children.

i Children c_i and c_{i+1} with degrees at least t and at most $2t-2$, respectively.

Output

Merged children $x.c_i$ and $x.c_{i+1}$.

Complexity

$O(t)$

```
procedure MoveKeyNext(x, i)
    a := x.c_i, b := x.c_{i+1}
    ▷ Move keys right to make room for the moving one.
    for j := 1 to b.n do
        b.k_{j+1} := b.k_j
        if not b.l then
            b.c_{j+1} := b.c_j
    b.n := b.n + 1
    b.k_1 := x.k_i
    x.k_i := a.k_{n+1}
    b.c_1 := a.c_{n+1}
    delete a.k_{n+1}
    delete a.c_{n+1}
    a.n := a.n - 1
    Write(x)
    Write(a)
    Write(b)
```

Algorithm 66 Moving a key

Input

x Node of the degree t.

i Children c_i and c_{i-1} with degrees at most $2t-2$ and at least t, respectively.

Output

Key from c_{i+1} moved to parent and parent key moved to c_i.

Complexity

$O(t)$

```
procedure MoveKeyPrev(x, i)
    a := x.c_i, b := x.c_{i-1}
    b.n := b.n + 1
    b.k_n := x.k_i
    x.k_i := a.k_1
    b.c_{n+1} := a.c_1
    ▷ Move keys left to fill empty slot.
    for j := 2 to a.n do
        a.k_{j-1} := a.k_j
        if not a.l then
            a.c_{j-1} := a.c_j
    delete a.k_{n+1}
    delete a.c_{n+1}
    a.n := a.n - 1
    Write(x)
    Write(a)
    Write(b)
```

Algorithm 67 Key inserting

Input

x Node where to insert the key.

K Key to insert into the given node x.

Output

Key K inserted.

Complexity

$O(\log n)$

```
procedure INSERT(x, K)
    i := x.n
    if x.l then
        ▷ Inserting into a leaf is putting the key to the proper position.
        while i ≥ 1 and K < x.k_i do
            x.k_{i+1} := x.k_i
            i := i − 1
            x.k_{i+1} := K
            x.n := x.n + 1
            Write(x)
    else
        while i ≥ 1 and K < x.k_i do
            i := i − 1
        i := i + 1
        Read(x.c_i)
        if x.c_i.n = 2 · t − 1 then
            Split(x, i)
            ▷ Key from c_i moved up to x, so check if K should be moved too.
            if K > x.k_i then
                i := i + 1
        Insert(x.c_i, K)
```

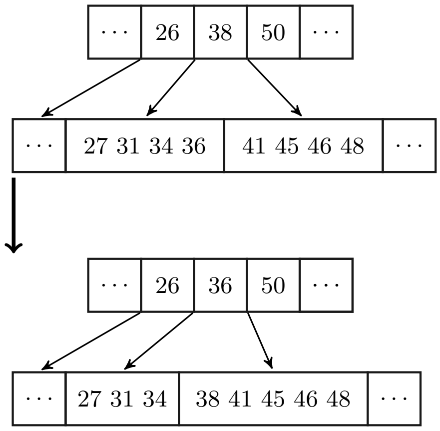

Figure 14: moving key 36

To insert key K into tree T, the algorithm starts at the root. If root is not full, use the above insert function directly. If not, create new root and split the original root.

Algorithm 68 Key inserting

Input

K Key to insert into the tree T.

Output

Key K inserted.

Complexity

$O(\log n)$

procedure INSERT(K)
  if $r.n = 2 \cdot t - 1$ **then**
    new s
    $r = s$
    $s.l =$ **false**
    $s.n := 0$
    $s.c_1 := r$
    Split$(s, 1)$
    Insert(s, K)
  else
    Insert(r, K)

10.4 Deleting

Deleting distinguishes cases on leaves and internal nodes. The following situations are possible for key K and subtree x:

D1 If the key K is in leaf x, then delete the key K from x.

D2 If the key K is in internal node x, then:

- **D2.1** If x's child y that precedes K has at least t keys, then delete the predecessor K' (which is placed in leaf of subtree y) of K and replace K by K' in x.
- **D2.2** Symmetrically, if x's child z that follows K has at least t keys, then delete the successor K' (which is stored in leaf of subtree z) of K and replace K by K' in x.
- **D2.3** Otherwise, if both y and z have only $t-1$ keys, merge K and all of z into y, so that x loses both K and the pointer to z, and y now contains $2t-1$ keys. Then, delete z and recursively delete K from y.

D3 If the key K is not present in internal node x, find child c_i that contains K. If c_i has only $t-1$ keys, execute step D3.1 or D3.2 as necessary to guarantee that we descend to a node containing at least t keys. Then, recursively delete K on c_i.

D3.1 If c_i has only $t-1$ keys but has an immediate sibling with at least t keys, move key from sibling to c_i.

D3.2 If c_i and both of c_i's immediate siblings have $t-1$ keys, merge c_i with one sibling.

Algorithm 69 Key deleting

Input

x Node where to delete the key.
K Key to delete.

Output

Key K deleted.

Complexity

$O(\log n)$

```
procedure DELETE(x, K)
    i := Index(K, x)
    if i ≠ null then ▷ cases D1 - D2
        if x.l then ▷ case D1
            for j := i to x.n + 1 do
                x.k_j = x.k_{j+1}
            delete x.k_{n+1}
            x.n := x.n − 1
            Write(x)
        else▷ case D2
            y := x.c_i, z := x.c_{i+1}
            if y.n ≥ t then ▷ (case D2.1)
                (a, j) := Predecessor(x, i)
                K' := a.k_j
                Delete(y, K) ▷ case D1
                x.k_i := K
                Write(x)
            else if z.n ≥ t then ▷ case D2.2
                (a, j) := Successor(x, i)
                K := a.k_j
                Delete(z, K) ▷ case D1
                x.k_i := K
                Write(x)
            else▷ case D2.3
                Merge(x, i) ▷ moves K from x to y
                Delete(y, K) ▷ case D3
```

10.5 Complexity

B tree with one, two or three elements has only one (root) node. B tree with four elements can have at most two nodes, having at least two elements in the child element. If node x has zero keys then it has one child.

Lemma 3. If $n \geq 1$ and $t \geq 2$, then for every tree with n nodes and degree t, height of the tree is not greater than $\log_t \frac{n+1}{2}$.

Algorithm 70 Key deleting

```
else▷ case D3
    i := IndexChild(K, x)
    if x.c_i.n = t − 1 then
        if 1 < i < x.n + 1 then
            if x.c_{i−1}.n ≥ t then ▷ case D3.1
                MoveKeyNext(x, i − 1)
            else if x.c_{i+1}.n ≥ t then ▷ case D3.1
                MoveKeyPrev(x, i + 1)
            else▷ case D3.2
                Merge(x, i)
        else if i = 1 then
            if x.c_{i+1}.n = t − 1 then ▷ (case 3.2)
                Merge(x, i)
            else▷ (case 3.1)
                MoveKeyPrev(x, i + 1)
        else if i = x.n + 1 then
            if x.c_{i−1}.n = t − 1 then ▷ case D3.2
                Merge(x, i − 1)
            else▷ case D3.1
                MoveKeyNext(x, i − 1)
        Delete(x.c_i, K)
    else
        Delete(x.c_i, K)
return x
```

Theorem 7. Complexity of find, insert and delete operations is $O(\log n)$.

Proof. Follows from lemma 3. QED

B tree* is a B tree where each node has at least $\frac{2}{3}$ full, i.e. contains at least $\frac{4}{3}t - 1$ keys. Inserting splits two full sibling nodes into three, so each of them is $\frac{2}{3}$ full. Since this scheme ensures that storage utilization is relatively high, height of B* tree is relatively smaller, consequently the find operation takes less time than in B tree.

Red black tree where each black node absorbs its red children is B tree. Such black node becomes node with three keys and four children at most.

11 B$^+$ tree

Motivation for B$^+$ tree is to have data structure with the properties as for B tree, while keys can be accessed in batches. Thus, for each key the adjacent keys can be found in constant time.

11.1 Definition

Definition 11.1. *B$^+$ tree* is B tree with the additional requirements:

1. All keys are stored in the leaves.
2. Leaves form a linked list starting from the leftmost leaf. It is called *sequence set*.
3. Internal nodes do not necessarily keep all of the keys. Those that are present, separate keys of the children in the same way as in B tree. They form so called *index*.

So, while B tree stores keys in all nodes (internal and leaves), B$^+$ tree keeps all keys in leaves and some of them in internal nodes. In addition, all leaves are linked into one single linked list.

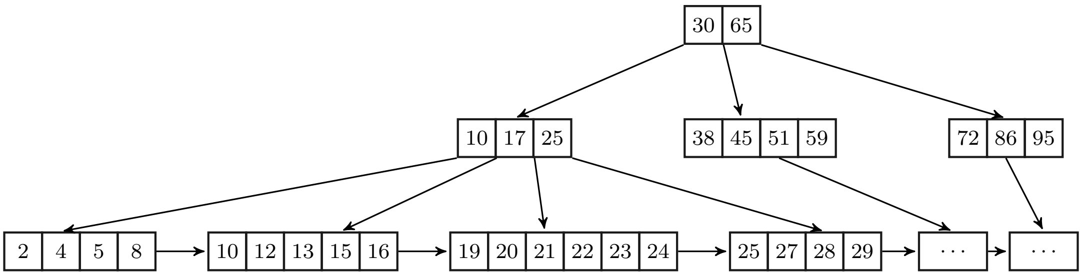

Figure 15: Example of B$^+$ tree of degree $t = 4$

The figure 15 shows an example of B$^+$ tree where 17 is not the key since it is present in an internal node only, while 10 and 25 are keys which also occur in internal nodes.

In the pseudo code, all notations remain same as for B tree. Additionally, each leaf x has a pointer $x.a$ to an adjacent leaf.

11.2 Searching

Starting from the root of a B$^+$ tree, the algorithm finds appropriate child as in the case of B tree. If key K is found in an internal node, then the search is not stopped, but the appropriate right pointer is chosen, so the algorithm proceeds down to a leaf.

From the algorithm follows that it doesn't matter which keys are stored in internal nodes, as long they separate keys in leaves in a proper way.

11.3 Auxiliary node operations

Splitting node is performed in a similar manner as in B tree. The difference is that central key is copied to a parent node and placed in the right sibling. Additionally, the sequence set is updated if necessary.

The split method is slightly modified to support copying of the central key to both parent and sibling node.

Merging is similar to the one on B tree except that central key is not copied from parent to the merged children. Additionally, the sequence set is updated if necessary.

Moving key $K \in c_i$ is performed in a manner similar to the B tree's move. K replaces the corresponding parent key which splits c_i and c_{i+1} and K is copied to c_{i+1} to be it's first key.

11.4 Insert

Inserting node is exactly the same as for B tree, except the modified split for B$^+$ tree is used.

Algorithm 71 Key finding

Input
  K Key to find.
  x Subtree where to look for K.
Output
  Node which contains K or null.
Complexity
  $O(\log n)$

```
procedure FIND(K, x)
    z := x
    while z ≠ null do
        i := 1
        while i ≤ z.n and K > z.k_i do
            i := i + 1
        if i ≤ z.n and K = z.k_i and z.l = true then
            break ▷ z found
        if z.l = true then
            z := null ▷ z not found
        else
            z := Read(z.c_i)
    returnz
```

11.5 Delete

Deleting key K is easier than in case of B tree, because all keys are in leaves. If K is in the index only, it is not deleted, because it keeps to separate keys in the index in a proper way. Thus, the following cases are distinguished:

D1 If the key K is in leaf x, then delete the key K from x.

D2 Find child c_i that contains K. If c_i has only $t-1$ keys, execute step D2.1 or D2.2 as necessary to guarantee that we descend to a node containing at least t keys. Then, recursively delete K on c_i.

- **D2.1** If c_i has only $t-1$ keys but has an immediate sibling with at least t keys, move key from sibling to c_i.
- **D2.2** If c_i and both of c_i's immediate siblings have $t-1$ keys, merge c_i with one sibling.

Algorithm 72 Key split

Input

x Node of degree t to split.

i Full child at this position

Output

None.

Complexity

$O(t)$

procedure SPLIT(x, i)

  $y := x.c_i$ ▷ Full node.
  new z
  $z.l := y.l$
  $z.n := T.d - 1$
  ▷ Copy second half of keys from y to z, including central key.
  for $j := 1$ **to** $T.d$ **do**
    $z.k_j := y.k_{T.d-1+j}$
  ▷ Copy second half of children from y to z, including central key.
  if not $y.l$ **then**
    for $j := 1$ **to** $T.d + 1$ **do**
      $z.c_j := y.c_{T.d-1+j}$
  $y.n := T.d - 1$
  ▷ Move x's children one place to the right to make room for z.
  for $j := x.n + 1$ **downto** $i + 1$ **do**
    $x_c j + 1 := x.c_j$
  $x.c_{i+1} := z$
  ▷ Add new key $y.k_t$ for z into x.
  for $j := x.n$ **downto** i **do**
    $x_k j + 1 := x.k_j$
  $x_i := y_t$
  $x.n := x.n + 1$
  ▷ Update sequence set if necessary.
  if $y.l =$ **true then**
    $z.a := y.a$
    $y.a := z.a$
  Write(x)
  Write(y)
  Write(z)

Algorithm 73 Key merge

Input

x Node to merge of degree t.

i Index of x's children c_i and c_{i+1} to merge.

Output

x's children on position i merged.

Complexity

$O(t)$

```
procedure MERGE(x, i)
    y := x.c_i, z := x.c_{i+1}
    ▷ Move i-th key of x into y.
    y.k_t := x.k_i
    ▷ Move the rest of x's keys to the left.
    for j := i to x.n do
        x.k_j := x.k_{j+1}
    delete x.k_{n+1}
    x.n := x.n − 1
    ▷ Copy z's keys into y.
    for j := 1 to T.d − 1 do
        y.k_{t+j} := z.k_j
    ▷ Copy z's children into y.
    if z.l then
        for j := 1 to t do
            y.c_{t+j} := z.c_j
    y.n = 2 · T.d − 1
    ▷ Update sequence set if necessary.
    if y.l then
        y.a := z.a
    delete z
    ▷ Remove link for z from x.
    delete x.c_{i+1}
    for j := i to x.n do
        x.c_j := x.c_{j+1}
    delete x.c_{n+1}
    Write(x)
    Write(y)
    Write(z)
```

Algorithm 74 Moving key to the next

Input

x Node of degree t with a key at i-th place.

i Index of x's children c_i and c_{i+1} with degrees at least t and at most $2t-2$, respectively.

Output

Key from c_i moved to parent and parent key moved to c_{i+1}.

Complexity

$O(t)$

```
procedure MoveKeyNext(x, i)
    a := x.c_i, b := x.c_{i+1}
    ▷ Move keys right to make room for the moving one.
    for j := 1 to b.n do
        b.k_{j+1} := b.k_j
        if not b.l then
            b.c_{j+1} := b.c_j
    b.n := b.n + 1
    b.k_1 := x.k_i := a.k_{n+1}
    b.c_1 := a.c_{n+1}
    delete a.k_{n+1}
    delete a.c_{n+1}
    a.n := a.n − 1
    Write(x)
    Write(a)
    Write(b)
```

Algorithm 75 Moving key to the previous

Input

x Node of degree t with a key at i-th place.

i Index of x's children c_i and c_{i-1} with degrees at most $2t-2$ and at least t, respectively.

Output

Key from c_{i-1} moved to parent and parent key moved to c_i.

Complexity

$O(t)$

```
procedure MoveKeyPrev(x, i)
    a := x.c_i, b := x.c_{i−1}
    b.n := b.n + 1
    b.k_n := x.k_i := a.k_1
    b.c_{n+1} := a.c_1
    ▷ Move keys left to fill empty slot.
    for j := 2 to a.n do
        a.k_{j−1} := a.k_j
        if not a.l then
            a.c_{j−1} := a.c_j
    delete a.k_{n+1}
    delete a.c_{n+1}
    a.n := a.n − 1
    Write(x)
    Write(a)
    Write(b)
```

Algorithm 76 Deleting key

Input
  x Subtree where to delete the key.
  K Key to delete.
Output
  Node from which the key is deleted.
Complexity
  $O(\log n)$

```
procedure DeleteKey(x, K)
    i := FindIndex(K, x)
    if i ≠ null then
        if x.l then ▷ Case D1.
            for j := i to x.n + 1 do
                x.k_j := x.k_{j+1}
            delete x.k_{n+1}
            x.n := x.n − 1
            Write(x)
    else ▷ Case D2.
        i := FindIndexChild(K, x)
        if x.c_i.n = T.d − 1 then
            if 1 < i < x.n + 1 then
                if x.c_{i−1}.n ≥ T.d then ▷ Case D2.1.
                    MoveKeyNext(x, i − 1)
                else if x.c_{i+1}.n ≥ T.d then ▷ Case D2.1.
                    MoveKeyPrev(x, i + 1)
                else ▷ Case D2.2.
                    Merge(x, i)
            else if i = 1 then
                if x.c_{i+1}.n = T.d − 1 then ▷ Case D2.2.
                    Merge(x, i)
                else ▷ Case D2.1.
                    MoveKeyPrev(x, i + 1)
            else if i = x.n + 1 then
                if x.c_{i−1}.n = T.d − 1 then ▷ Case D2.2.
                    Merge(x, i − 1)
                else ▷ Case D2.1.
                    MoveKeyNext(x, i − 1)
            Delete(x.c_i, K)
        else
            Delete(x.c_i, K)
    return x
```

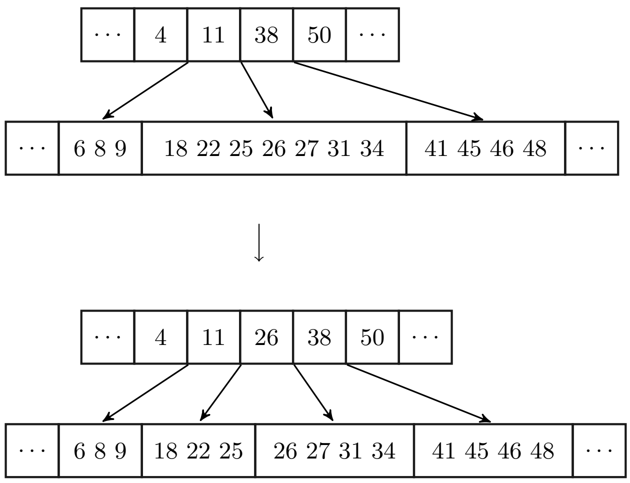

Figure 16: splitting 7-elements node ($t = 4$), key 26 is copied to the parent

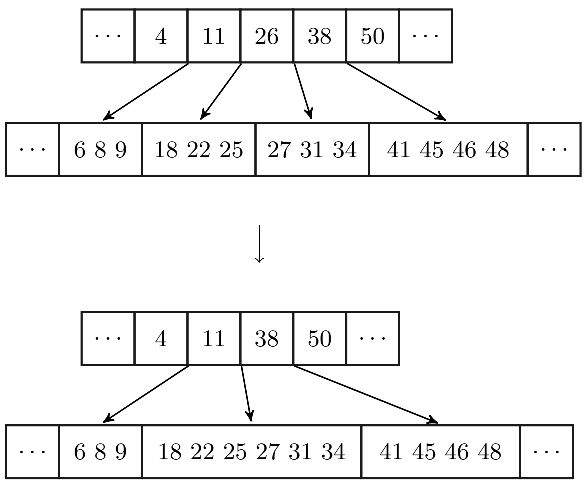

Figure 17: merging two 3-elements nodes ($t = 4$)

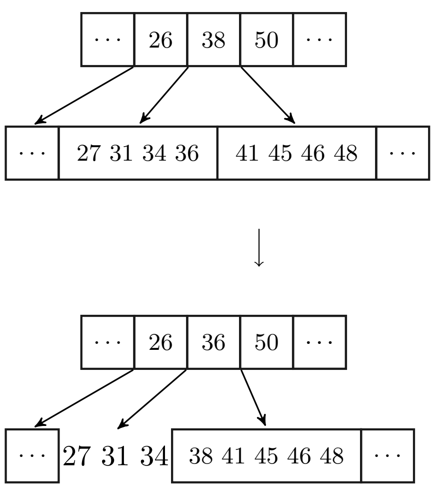

Figure 18: moving key 36

12 Splay tree

The motivation is to have a binary tree which performs rotations when a node is accessed. That way, frequently used nodes are moved up to tree and faster retrieved, which is needed by LRU cache. Operations of interest are finding, inserting and deleting key, joining two trees and splitting a tree into at given key into two subtrees.

Let T be a binary tree with root r_T; for each node $x \in T$ let p, p' be the pointers to its left, right children, parent and grand parent, respectively. *Splaying* tree T of size n at node x is sequence of the following *splay steps* until x becomes the root of T:

zig If p is the root, rotate the edge (x, p), and finish the sequence.

zig-zig If p is not the root and x and p are both left/right children of their respective parents, rotate the edge (p, p') and then rotate (x, p).

zig-zag If p is not the root, x is left child of p and p is right child of p', or vice versa (x is right child of p and p is left child of p'), then rotate (x, p) and then rotate x with the new p (which was p' before this step).

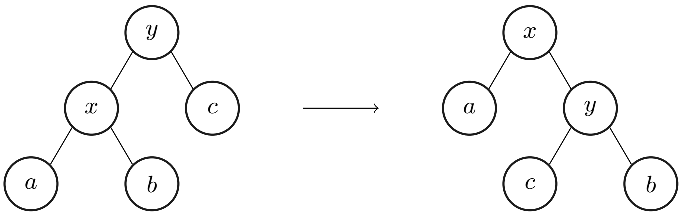

Figure 19: Zig operation on x

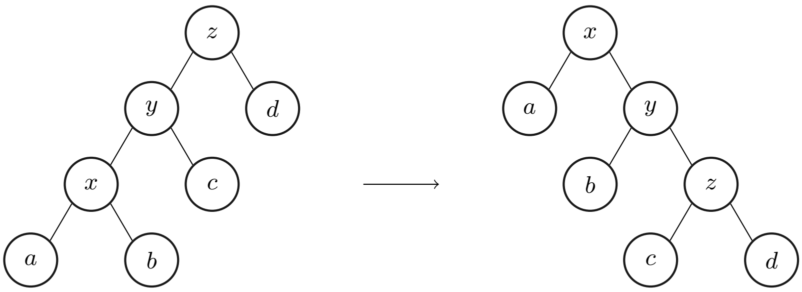

Figure 20: Zig-zig operation on x

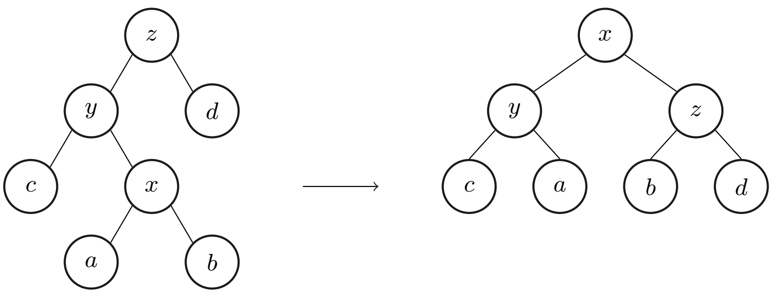

Figure 21: Zig-zag operation on x

Binary search tree T is *splay tree* if for each operation (searching, inserting, deleting) on node x an additional splaying on x is performed.

12.1 Rotation, linking, assembling

Rotations of nodes are performed as for AVL trees.

If a key K is searched in a tree T, then few subtrees can be recognized. *Left tree* L is a tree containing all nodes from T less than K. *Right tree* R is defined similarly. *Middle tree* is a subtree of T rooted at the current node reached during the search.

If node x is right child of its parent, *left linking* moves x to be the most right child of L (so it becomes $\max(L)$). If node x is left child of its parent, *right linking* moves x to be the most left child of R (so it becomes $\min(R)$).

For a left and right trees L and R, and node x with left and right children a, b, *assembling* creates single tree such that $x.c_l = L, x.c_r = R, L.c_r = a, R.c_l = b$.

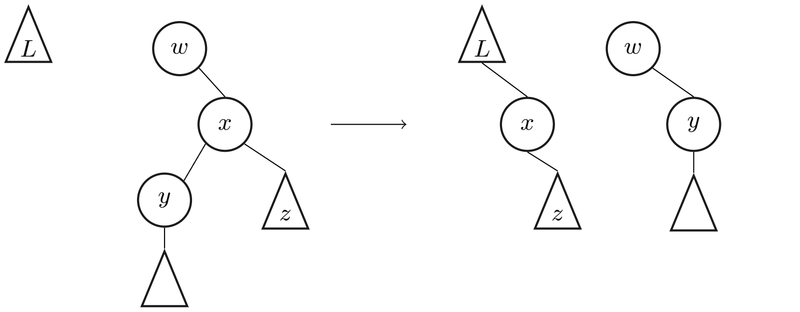

Figure 22: Left linking of node x

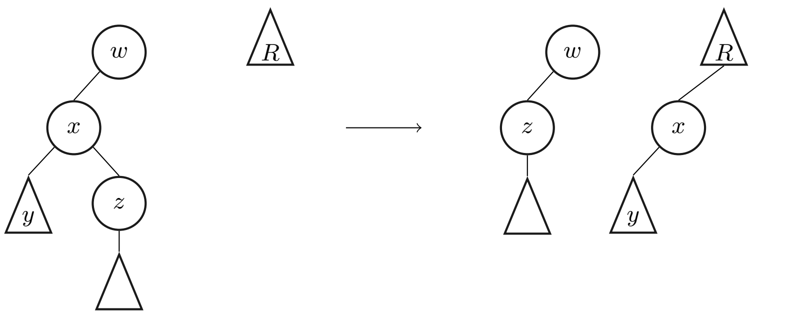

Figure 23: Right linking of node x

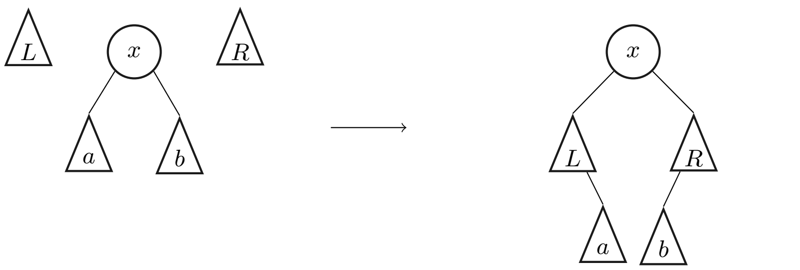

Figure 24: Assembling of node x

12.2 Splaying

Splaying as defined at the start assumes that it starts from a node x and goes until root is reached. That's bottom-up splaying; it's appropriate when direct access to the node is available.

All three $x = x^p{}_l$ cases perform right rotation. If right rotation is done by default, then the remaining left rotation of zig-zag case can be delayed to the zig step of $x = x^p{}_r$ case. So, the function above can be simplified:

Algorithm 77 Left linking of a node.

Input

L Tree to link to.

x Node to link with the tree L.

Output

x left linked to L.

procedure LINKLEFT(L, x)

$y := x.c_l$

$w := x.p$

$y.p := w$

$w.c_r := y$

$x.p := L$

$L.c_r := x$

Algorithm 78 Right linking of a node.

Input

R Tree to link to.

x Node to link with the tree R.

Output

x left linked to L.

procedure LINKRIGHT(R, x)

$z := x.c_r$

$w := x.p$

$z.p := w$

$w.c_l := z$

$x.p := R$

$R.c_l := x$

Algorithm 79 Assembling of a node.

Input

L Left tree to assemble.

R Right tree to assemble.

x Node to assemble with L and R.

Output

x, L, R assembled.

procedure ASSEMBLE(x, L, R)

$a := x.c_l, b := x.c_r$

$x.c_l := L, x.c_r := R$

$L.p := x, R.p := x$

$L.c_r := a, R.c_l := b$

$a.p := L, b.p := R$

Algorithm 80 Splaying in the bottom-up manner.

```
Input
    x   Node to splay.
Output
    Tree splayed at x.
procedure SplayUp1(x)
    while x.p ≠ null do
        if x = x.p.c_l then
            if x.p.p = null then ▷ Zig
                RotateRight(x.p)
            else if x.p = x.p.p.c_l then ▷ Zig-zig
                RotateRight(x.p.p)
                RotateRight(x.p)
            else if x.p = x.p.p.c_r then ▷ Zig-zag
                RotateRight(x.p)
                RotateLeft(x.p)
        else if x = x.p.c_r then
            if x.p.p = null then ▷ Zig
                RotateLeft(x.p)
            else if x.p = x.p.p.c_r then ▷ Zig-zig
                RotateLeft(x.p.p)
                RotateLeft(x.p)
            else if x.p = x.p.p.c_l then ▷ Zig-zag
                RotateLeft(x.p)
                RotateRight(x.p)
```

Algorithm 81 Splaying in the bottom-up manner simplified.

```
Input
    x   Node to splay.
Output
    Tree splayed at x.
procedure SplayUp2(x)
    while x.p ≠ null do
        if x = x.p.c_l then
            if x.p = x.p.p.c_l then
                RotateRight(x.p.p)
            RotateRight(x.p)
        else if x = x.p.c_r then
            if x.p = x.p.p.c_r then
                RotateLeft(x.p.p)
            RotateLeft(x.p)
```

In some cases, splaying during traversing the tree T can be performed in a more efficient way. Top-down splaying starts from a node x and executes splaying steps until a node t with key K is reached (which is the accessed node). All accessed nodes on the path are classified into left or right subtree; x is moving toward T's bottom by classifying accessed nodes into left and right tree. When x becomes t, middle tree rooted at x is assembled with L and R.

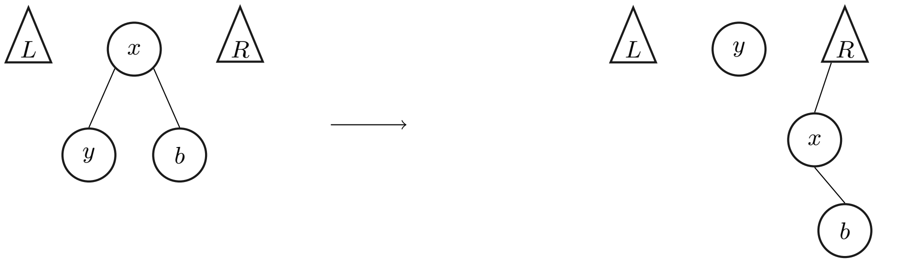

Figure 25: Splay down of x for case R1

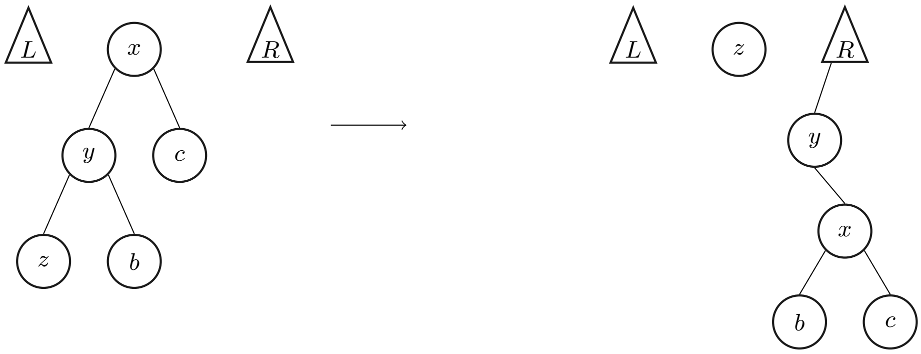

Figure 26: Splay down of x for case R2

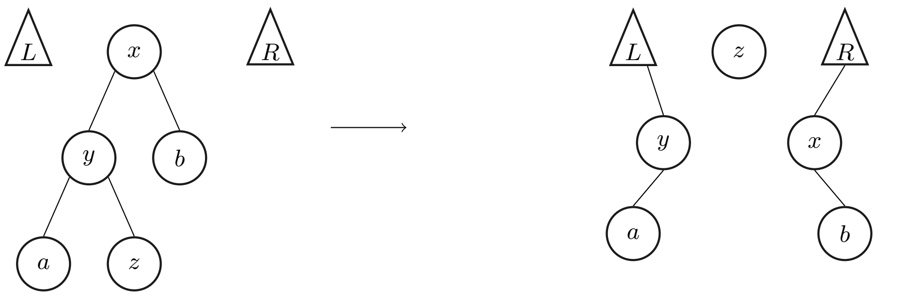

Figure 27: Splay down of x for case R3

12.3 Finding key

Finding a key K in subtree T goes by traversing the tree from the root choosing left or right subtree depending of the key in the current node (same as in binary search tree). If such node x is found, then splaying on x is performed. If there is no node x such that $x.k = K$, then the last (non-null) node on the search path is splayed (in the bottom-up manner). If T is empty, then splaying is not performed.

12.4 Joining trees

Assuming that tree T_1 is less than tree T_2 (i.e. all keys from T_1 are less than each key from T_2), *join* constructs a single tree of all items from T_1 and T_2. Join finds the largest key $m = \max(T_1)$ (by taking the rightmost node in T_1). Since finding m splays it (and makes $m = r_1$), the root r_1 has null right child. The operation is completed by making T_2 the right subtree of newly created root.

Algorithm 82 Splaying in the top-down manner.

```
Input
    x   Node to start splaying.
    K   Key until the splaying is performed.
Output
    Tree splayed at x.
procedure SplayDown(K, x)
    L := R := null
    while K ≠ x.k do
        if K < x.k then
            y := x.c_l
            if K = y.k then ▷ Case R1
                LinkRight(R, x)
                x := y
            else if K < y.k then ▷ Case R2.
                z := y.c_l
                RotateRight(x)
                LinkRight(R, y)
                x := z
            else if K > y.k then ▷ Case R3.
                z := y.c_r
                LinkRight(R, x)
                LinkLeft(L, y)
                x := z
        else
            y := x.c_r
            if K = y.k then ▷ Case L1.
                LinkLeft(L, x)
                x := y
            else if K > y.k then ▷ Case L2.
                z := y.c_r
                RotateLeft(x)
                LinkLeft(L, y)
                x := z
            else if K < y.k then ▷ Case L3.
                z := y.c_l
                LinkLeft(L, x)
                LinkRight(R, y)
                x := z
    assemble(x, L, R)
```

Algorithm 83 Finding a key.

Input

K Key to find in a subtree.

Output

Node with key K or null (if K is not in T or subtree is empty).

Complexity

$O(\lg n)$

procedure Find(K)
  $x := r_T$
  if $x =$ **null then**
    return null
  while $x \neq$ **null do**
    $y := x$ ▷ Track parent of x.
    if $K < x.k$ **then**
      $x = x.c_l$
    else if $K > x.k$ **then**
      $x := x.c_r$
    else
      break
  if $x \neq$ **null then**
    SplayUp(x)
  else
    SplayUp(y)
  return x

Algorithm 84 Joining trees.

Input

T_1 First tree to join.

T_2 Second tree to join.

Output

Node with key K or null (if K is not in T or subtree is empty).

Complexity

$O(\lg n)$

procedure Join(T_1, T_2)
  $m := \max(T_1)$
  Splay(m)
  $m.c_r := T_2$
  $r_2.p := m$

12.5 Splitting a tree

Splitting tree T at key K returns two subtrees T_1 and T_2 by breaking T at the node which contains K. The operation is accomplished by finding node with K and then returning two trees formed by breaking left or right link of the new root. If K is not in T, then the last non-null node found during the search will be used for splitting.

Algorithm 85 Splitting a tree.

Input
K Key to split on.
Output
trees T_1, T_2 obtained by splitting.
Complexity
$O(\lg n)$

procedure SPLIT(K)
 Find(K) ▷ Moves K to root.
 $T_1 := r_T$
 $T_2 := r_T.c_r$
 $r_T.c_r :=$ **null**
 $r_{T_2}.p :=$ **null**
 return (T_1, T_2)

12.6 Inserting key

Inserting key K goes by splitting T on K (which returns two subtrees T_1 and T_2), and then replacing T with a new root containing K and left/right subtrees T_1 and T_2. Since K does not exist in T, splitting is performed on the last non-null node found during the search.

Algorithm 86 Splitting a tree.

Input
K Key to insert into tree T.
Output
T Tree with the key K.
Complexity
$O(\lg n)$

procedure INSERT(K)
 $(T_1, T_2) :=$ Split(K)
 $r_T.k := K$
 $r_T.c_l := T_1, T_1.p := r_T$
 $r_T.c_r := T_2, T_2.p := r_T$

12.7 Deleting a key

Deleting key K goes by finding K (splaying moves it to root) and then replacing T by joining $r_T.c_l$ and $r_T.c_r$.

12.8 Amortized complexity

For each $x \in T$ of splay tree T, define it's size as $s(x) = \sum_{y \in T_x} w(y)$, where T_x is subtree rooted at x. Then, let's define rank of x as $\lambda(x) = \log s(x)$. Finally, define potential of T as $\Phi(T) = \sum_{x \in T} \lambda(x)$.

Algorithm 87 Deleting a key.

Input
K Key to delete from tree T.
Output
T Tree without the key K.
Complexity
$O(\lg n)$

procedure DELETE(K)
  Find(K)
  join($r_T.c_l, r_T.c_r$)

Lemma 4. The amortized time to splay tree with root $r(T)$ at node x is

$$\hat{c}_x = 3(\lambda(r(T)) - \lambda(x)) + 1 = O\left(\log \frac{s(r(T))}{s(x)}\right)$$

Proof. See [5] for the proof. QED

For a sequence of m accesses on n-node splay tree, the potential decrease is $\Phi_m - \Phi_0$ where

$$\Phi_0 = \sum_{i=1}^{n} \lambda(i) = \sum_{i=1}^{n} \log s(i) = \sum_{i=1}^{n} \log \left(\sum_{j \in T_i} w(j)\right) \leq \sum_{i=1}^{n} \log \left(\sum_{j=1}^{n} w(j)\right)$$

$$\Phi_m = \sum_{i=1}^{n} \log \left(\sum_{j \in T_i^m} w(j)\right) \geq \sum_{i=1}^{n} \log w(i)$$

with T_i as subtree at i-th node before splaying and T_i^m as subtree at i-th node after m-th splay operation. Thus,

$$\Phi_m - \Phi_0 \leq \sum_{i=1}^{n} \log W - \sum_{i=1}^{n} \log w(i) = \sum_{i=1}^{n} \log \frac{W}{w(i)}$$

where $W = \sum_{i=1}^{n} w(i)$.

Let's apply the obtained results to the tree operations. For $x \in T$ denote with x_p and x_s predecessor and successor of x.

Theorem 8. The amortized time of search operation for $x = find(K)$ is

$$\begin{cases} 3 \log \frac{W}{w(x)} + 1, x \in T \\ 3 \log \frac{W}{min\{w(x_p), w(x_s)\}} + 1, x \notin T \end{cases}$$

Proof. If $x \in T$, by lemma 4, the splaying time for node x is $3 \log \frac{s(r(T))}{s(x)} + 1 \leq 3 \log \frac{W}{w(x)} + 1$, where $W = s(r(T)), s(x) \geq w(x)$. If $x \notin T$, then either x_p or x_s is in T, so $s(x) \geq \min\{w(x_p), w(x_s)\}$ and thus splaying time is $3 \log \frac{W}{\min\{w(x_p), w(x_s)\}} + 1$. QED

Theorem 9. The amortized time of join operation is

$$3 \log \frac{W}{w(x)} + O(1)$$

where $x = \max T_1$.

Proof. The bound on join is immediate from the bound on find - the splaying time is at most $3 \log \frac{s(T_1)}{w(x)} + 1$. The increase in potential caused by linking T_1 and T_2 is $\log \frac{s(T_1)+s(T_2)}{s(T_1)} \leq 3 \log \frac{W}{s(T_1)}$ (because $W = s(T_1) + s(T_2)$). QED

Theorem 10. The amortized time of split operation is

$$\begin{cases} 3\log\frac{W}{w(x)} + O(1), x \in T \\ 3\log\frac{W}{min\{w(x_p), w(x_s)\}} + O(1), x \notin T \end{cases}$$

where x is node such that $k(x) = K$.

Proof. Since the searching operation is the only non-constant time operation, then amortized time of split is same as of find. QED

Theorem 11. The amortized time of insert operation is

$$3\log\frac{W - w(x)}{\min\{w(x_p), w(x_s)\}} + \log\frac{W}{w(x)} + O(1)$$

where x is such that $k(x) = K$.

Proof. Follows from the complexity of finding a key. QED

Theorem 12. The amortized time od delete operation is

$$3\log\frac{W}{w(x)} + 3\log\frac{W - w(x)}{w(x_p)} + O(1)$$

Proof. Result follows from the bounds on find and join operations. QED

Theorem 13. The amortized time of search operation for $x = find(K)$ is $O(\lg n)$, where $n = |T|$ is number of nodes in T.

Proof. Assigning $w(x) = 1/n$ in theorem 8 the proof follows. QED

12.9 Few theorems

By using lemma 4 various corollaries can be obtained.

Theorem 14 (Balance theorem)**.** For a sequence of m operations on n-node tree the total access time is $O((m + n)\log n + m)$.

Proof. Assign weight $w(i) = 1/n$ for each node $i = 1, \dots n$. Then, $W = \sum_{i=1}^{n} w(i) = \sum_{i=1}^{n} 1/n = 1$. Since $s(r(T)) = W = 1$, the amortized access is

$$a_j = 3\log\frac{s(r(T))}{s(j)} + 1 = 3\log\frac{1}{\sum_{i\in T_j} w(i)} + 1 \le 3\log\frac{1}{w(i)} + 1 = 3\log n + 1$$

so the potential decrease is

$$\Phi_m - \Phi_0 \le \sum_{i=1}^{n} \log\frac{W}{w(i)} = \sum_{i=1}^{n} \log n = n\log n$$

Thus, total access time is

$$\sum_{j=1}^{m} t_j = \sum_{j=1}^{m} a_j + n\log n = \sum_{j=1}^{m} (3\log n + 1) + n\log n = 3m\log n + m + n\log n =$$

$$(3m + n)\log n + m = O\left((m + n)\log n + m\right)$$

QED

For node i let $q(i) \ge 1$ be it's access frequency, i.e. the total number of times i is accessed. For sequence of m accesses it would be $\sum_{i=1}^{n} q(i) = m$, and thus $n \le m$.

Theorem 15 (Static Optimality Theorem)**.** If every node is accessed at least once, then the total access time is

$$O\left(m + \sum_{i=1}^{m} q(i) \log \frac{m}{q(i)}\right)$$

Proof. Assign $w(i) = q(i)/m, i = 1, \dots n$. Then, $s(r(T)) = W = \sum_{i=1}^{n} w(i) = 1$. Since $s(i) \geq w(i) = q(i)/m$, then

$$a_i = 3 \log \frac{s(r_T)}{s(i)} + 1 = 3 \log \frac{1}{\frac{q(i)}{m}} + 1 = 3 \log \frac{m}{q(i)} + 1$$

and the potential decrease over the sequence is

$$\Phi_m - \Phi_0 = O\left(\sum_{i=1}^{n} \log \frac{W}{w(i)}\right) = O\left(\sum_{i=1}^{n} \log \frac{m}{q(i)}\right)$$

Thus,

$$\sum_{j=1}^{m} t_j = \sum_{j=1}^{m} \left(3 \log \frac{m}{q(j)} + 1\right) + O\left(\sum_{i=1}^{n} \log \frac{m}{q(i)}\right) =$$

$$O\left(m + \sum_{j=1}^{m} \log \frac{m}{q(i)}\right) + O\left(\sum_{i=1}^{n} \log \frac{m}{q(i)}\right) = O\left(m + \sum_{j=1}^{m} \log \frac{m}{q(i)}\right) + O\left(\sum_{i=1}^{m} \log \frac{m}{q(i)}\right) =$$

$$O\left(m + \sum_{i=1}^{m} \log \frac{m}{q(i)}\right) = O\left(m + \sum_{i=1}^{m} q(i) \log \frac{m}{q(i)}\right)$$

QED

Assume that nodes are numbered from 1 to n in symmetric order and the sequence of accessed nodes is $i_1, \dots i_m$.

Theorem 16 (Static Finger Theorem)**.** If f is any fixed node, the total access time is $O(n \log n + m + \sum_{j=1}^{m} \log(|i_j - f| + 1))$

Proof. Assign $w(i) = 1/(|i - f| + 1)^2$ to each node i. Then,

$$W = \sum_{i=1}^{n} w(i) = \sum_{i=1}^{n} \frac{1}{(|i - f| + 1)^2} \leq 2 \sum_{k=1}^{\infty} \frac{1}{k^2} = O(1)$$

$$a_j = 3\left(\log \frac{W}{s(j)}\right) \leq 3\left(\log \frac{W}{w(j)}\right) = O\left(\log \frac{1}{\frac{1}{(|i_j - f| + 1)^2}}\right) = O\left(2 \log(|i_j - f| + 1)\right) = O\left(\log(|i_j - f| + 1)\right)$$

$$\Phi_m - \Phi_0 = O\left(\sum_{i=1}^{m} \log \frac{W}{w(i)}\right) = O\left(\sum_{i=1}^{m} \log \frac{1}{\frac{1}{(|i-f|+1)^2}}\right) = O\left(\sum_{i=1}^{n} \log(|i - f| + 1)^2\right) = O(n \log n)$$

so the total access time is

$$\sum_{j=1}^{m} t_j = \sum_{j=1}^{m} O\left(\log(|i_j - f| + 1)\right) + O(n \log n) = O\left(n \log n + \sum_{j=1}^{m} \log(|i_j - f| + 1)\right) =$$

$$O\left(n \log n + m + \sum_{j=1}^{m} \log(|i_j - f| + 1)\right)$$

QED

By changing the node weights as the accesses take place, interesting results can be obtained. Number the accesses from 1 to m in the order they occur. For any access j, let $t(j)$ be the number of different nodes accessed before access j since the last access of node i_j, or since beginning of the sequence if j is the first of node i_j.

Theorem 17 (Working Set Theorem)**.** Total access time is $O\left(n \log n + m + \sum_{j=1}^{m} \log\left(t(j)+1\right)\right)$.

Proof. Assign weights $1, 1/4, 1/9, \ldots, 1/n^2$ to the nodes in order by first access. Suppose that before access j node i_j has weight $w(i_j) = 1/k^2$. After the access j set $w(i_j) = 1$ and for each node i having $w(i) = 1/(k')^2, k' < k$, assign $w(i) = 1/(k'+1)^2$. Such reassignment permutes weights $1, 1/4, 1/9, \ldots, 1/n^2$ among the nodes and guarantees that $w(i_j) = \frac{1}{(t(j)+1)^2}$ during access j.

Since $W = \sum_{k=1}^{n} \frac{1}{k^2} = O(1)$, then $a_j = O(\log(t(j)+1))$. The weight reassignment after an access j increases the weight of the root, because $w(i_j) = 1$ and node i is moved to root (due to the splaying operations characteristics); weights of other nodes are decreased. The size of the root is unchanged, but the sizes of other nodes can decrease. Thus, potential $\Phi = \sum_{i=1}^{n} \lambda(i)$ can decrease on weights reassignment. Similarly, amortized time for weight reassignment after access j is not greater than zero:

$$a_i = 3 \log \frac{s(r_T)}{s(i)} + 1, a_i' = 3 \log \frac{s(r(T))}{s(i_j)} + 1$$

so

$$a_i' - a_i = 3 \log \frac{s(i)}{s(i_j)} \leq 0$$

since $s(i_j) > s(i)$ after access j (node i becomes the root). QED

Theorem 18 (Unified Theorem)**.** Total time of a sequence of m accesses on an n-node splay tree is

$$O\left(n \log n + m + \sum_{j=1}^{m} \log \min\left\{\frac{m}{q(i_j)}, |i_j - f| + 1, t(j) + 1\right\}\right)$$

where f is any fixed item.

13 Trie

Motivation for this data structure is to enable fast retrieval of strings and their common prefixes. Operations of interest are: finding, inserting and deleting key. The application of this data structure are the associative array, lexicographic sorting and the radix sort.

13.1 Definition

Trie (also known as prefix tree or digital tree) is a tree T defined over alphabet L with the following properties:

1. There is one root r_T only.
2. Each node x has arbitrary number of children determined by an array $x.c[i]$, where $i \in L$. If x is leaf, then $x.c$ is empty array i.e. it's length $|x.c|$ is zero.
3. For each child $x.c[i]$ there is a character $x.p[i] \in L$ which determines prefix for x.
4. All leaves and some of internal nodes $x \in T$ have associated values $x.v$. Position of x defines a key associated with it by appending all characters on the path from the root to x.

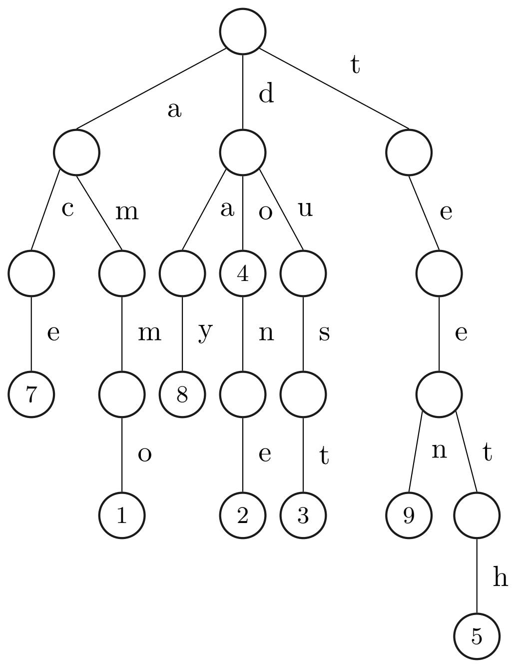

Figure 28: Trie with keys/values: ace/7, ammo/1, day/8, do/4, done/2, dust/3, teen/9, teeth/5

13.2 Search

The operation checks if the given key K exists in a trie T. It goes one by one character of K until the corresponding child exists. If all characters are traversed, then the key K is found; otherwise, the key does not exist.

13.3 Insertion

The operation puts a key/value pair (K, V) into trie T. It goes one by one character of K and checks whether they exist from the root down to leaves. If a character of K is not found on the path, it is added, as well all remaining characters. If all characters of K exist on the path, then the reached node is updated with value V.

Algorithm 88 Finding a key in a trie.

Input

K String to check for existence in a trie T.

Output

Value if exists or null if no such key is present.

Complexity

$O(|K|)$

```
procedure FIND(K)
    x := r_T
    for i := 1 to |K| do
        h := K[i] ▷ Current character.
        if x.p[h] ≠ null then
            x := x.c[h]
        else
            return null
    return x.v
```

Algorithm 89 Inserting a key/value into a trie.

Input

K String key to insert into T.

V Value to insert into T.

Output

T with the added (K, V).

Complexity

$O(|K|)$ is the worst case complexity.

```
procedure INSERT(K, V)
    if r_T = null then
        new r_T
    p_x := x := r_T
    for i := 1 to |K| do
        h := K[i]
        x := x.c[h]
        if x = null then
            break
        p_x := x
    while i <= |K| do
        h := K[i]
        new x
        p_x.c[h] := x
        new p_x.p[h]
        p_x := x
        i := i + 1
    x.v := V
```

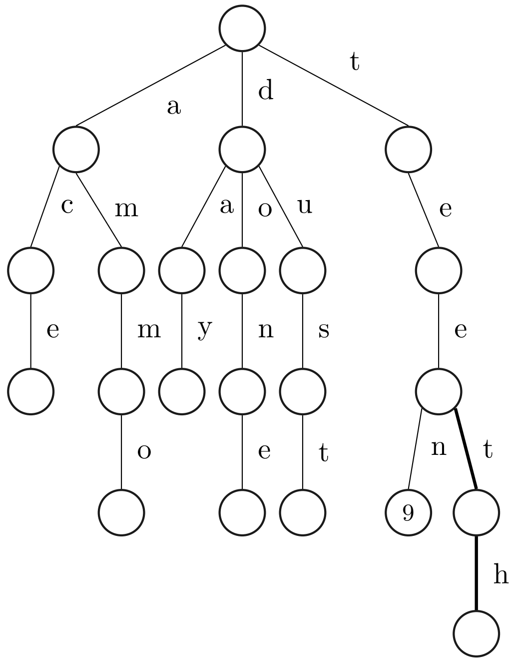

Figure 29: Inserting *teeth* into trie; bold edges are newly created

13.4 Deletion

Deleting key K finds the key in a trie T by traversing a path p from the root down to a node x that contains K. If x is a leaf, then all nodes on p which are single child are deleted. If x is not leaf, its value is dropped.

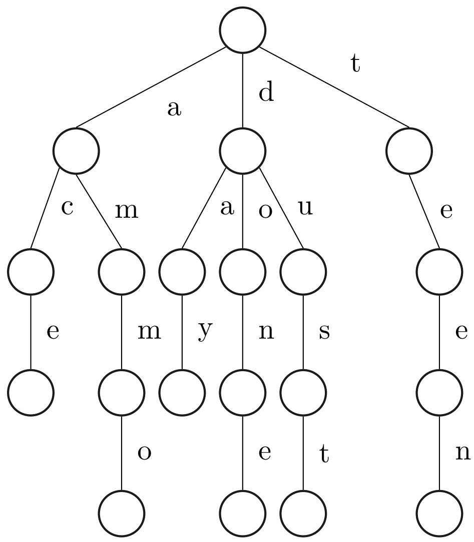

Figure 30: Trie after deleting the key *teeth*.

13.5 Worst case complexity

Theorem 19. For a trie T and a key K, finding, inserting and deleting the key have complexity $O(|K|)$,

Proof. Find has one loop of size $|K|$. Insert has two loops which in total are of size $|K|$. Delete has two loops: the first is obviously of size $|K|$, the second goes over a path which contains at most $|K|$ nodes. Thus, complexity of all operations is $O(|K|)$. QED

Algorithm 90 Deleting a key in a trie.

Input
  K Key to delete in the trie T.
Output
  T Trie without key K and true returned, false if no such key is present.
Complexity
  $O(|K|)$

```
procedure DELETE(K)
    if r_T = null then
        return false
    new P ▷ Stack of nodes traversed on the path of K.
    x := r_T
    for i := 1 to |K| do
        h := K[i]
        if x.p[h] ≠ null then
            x := x.p[h]
            P.push(x)
        else
            return false
    delete x
    ▷ Go up along the path and delete nodes which are the single child.
    i := |K|
    x := P.pop()
    while not P.empty() and |x.c| = 0 and i > 0 do
        delete x
        h := K[i]
        p_x := P.pop()
        if p_x ≠ null then
            delete p_x.p[h]
        x := p_x
        i := i − 1
```

14 Radix tree

Radix tree is a trie such that each node which is the only child of its parent is merged to its parent. Motivation for this data structure is to optimize space usage of nodes, so there are no nodes with only one child.

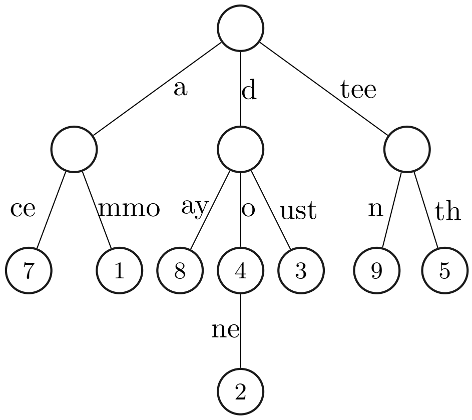

Figure 31: Radix tree with the keys/values: ace/7, ammo/1, day/8, do/4, done/2, dust/3, teen/9, teeth/5.

14.1 Search

For a given key K, start from the root by finding node x with a key that matches K's prefix. While there is such node, proceed with the procedure on x's children.

Algorithm 91 Finding a key in a radix tree.

Input
  K String K to find in radix tree T.
Output
  Value if exists or null if no such key is present.
Complexity
  $O(|K|)$

procedure FIND(K)
  $x := r_T$
  $L := 0$ ▷ Length.
  $f :=$ **true** ▷ Is found.
  while $L \leq |K|$ **and** $f =$ **true do**
    $f := \ False$
    for $i := 1$ **to** $x.s$ **do**
      $k' :=$ Substring$(K, L+1, x.k[i].l)$
      if $k' = x.k[i]$ **then**
        $x := x.c[k]$
        $L := L + k.l$
        $f :=$ **true**
        break
  if $f =$ **true then**
    return $x.v$
  else
    return null

14.2 Insert

Inserting key/value pair (K, V) into radix tree T goes by finding corresponding nodes which match a prefix of K. The rest of K (if any) is put into T.

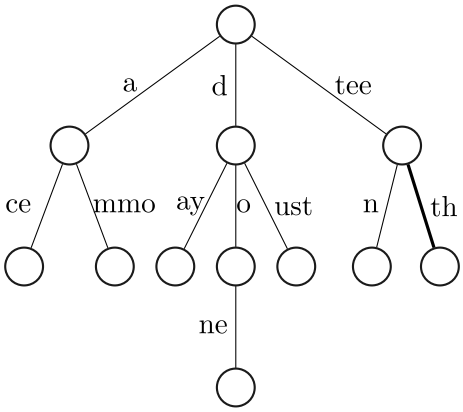

Figure 32: Inserting *teeth* into the radix tree; the bold edge is created.

14.3 Delete

To delete key K in radix tree T, find a corresponding node x for the key K; let p_x be x's parent. If x is a leaf, then it is deleted. In case that p_x after deletion of x remains with only one child y, then y's key is appended to p_x's.

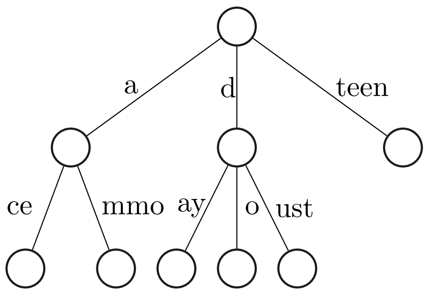

Figure 33: Radix tree after deleting the key *teeth*.

14.4 Worst case complexity

Theorem 20. For a trie T and a key K, finding, inserting and deleting the key have complexity $O(|K|)$,

Proof. All operations have loops of size $|K|$, thus their complexity is $O(|K|)$. QED

Algorithm 92 Inserting a key into a radix tree.

Input

K Key to insert into a radix tree T.

V Value to insert into a radix tree T.

Output

T Radix tree with the added key/value.

Complexity

$O(|K|)$

```
procedure INSERT(K, V)
    ▷ Find path that matches K's prefix.
    p_x := x := r_T
    L := 0 ▷ Length.
    f := true ▷ Is found.
    while L ≤ |K| and f = true do
        f := false
        for i := 1 to x.s do
            k' := Substring(K, L + 1, |x.k[i]|)
            if k' = x.k[i] then
                p_x := x
                x := x.c[k']
                L := L + |k'|
                f := true
                break
    ▷ If K's suffix which did not match existing keys exists, add it to a new child.
    if L < |K| then
        new x
        k' := Substring(K, L + 1, |K| − L)
        p_x.c[k] := x
        p_x.k[p_x.x + 1] := k'
        p_x.s := p_x.s + 1
    x.v := V
```

Algorithm 93 Deleting a key from a radix tree.

Input
K Key to delete from a radix tree T.
Output
T Radix tree without the deleted key.
Complexity
$O(|K|)$

procedure DELETE(K)
    $p_x := x := r_T$
    $k_{p_x} := k_x :=$ **null**
    $i_x := 0$
    $L := 0$
    $f :=$ **true**
    while $L < |K|$ **and** $f =$ **true do**
        $f :=$ **false**
        $k_{p_x} := k_x$
        for $i := 1$ **to** $x.s$ **do**
            $k' :=$ Substring$(K, L + 1, |x.k[i]|)$
            if $k' = x.k[i]$ **then**
                $p_x := x$
                $x := x.c[k']$
                $k_x := k', i_x := i$
                $L := L + |k'|$
                $f :=$ **true**
                break
    $x.v :=$ **null**
    ▷ In case the key from a leaf is deleted, remove the leaf.
    if $k' = |K|$ **then**
        delete x
        delete $p_x.c[k_x]$
        delete $p_x.k[i_x]$
        $p_x.s := p_x.s - 1$
        ▷ In case single child remains, concatenate key with the parent key.
        if $p_x.s = 1$ **then**
            $k_y := p_x.k[1]$
            $y := p_x.c[k_y]$
            $k_{p_x} := k_{p_x} + k_y$
            delete $p_x.k[1]$
            delete $p_x.c[1]$
            delete y
            $p_x. := 0$

15 Treap

A treap is a binary structure which keeps both randomized binary search tree and heap properties. That said, T is *treap* if

1. Every node x consists of a pair: key $x.k$ and priority $x.l$.
2. The binary search tree property holds: $\forall x \in T : x.c_l.k \leq x.c_r.k$.
3. The heap property holds: $\forall x \in T : x.p.l \leq x.l$.

Construction of a binary tree could lead to various tree shapes: it can be a list or perfectly balanced. The shape is determined by the random permutation of the numbers $(1, 2, 3, 4, 5, 6, 7)$.

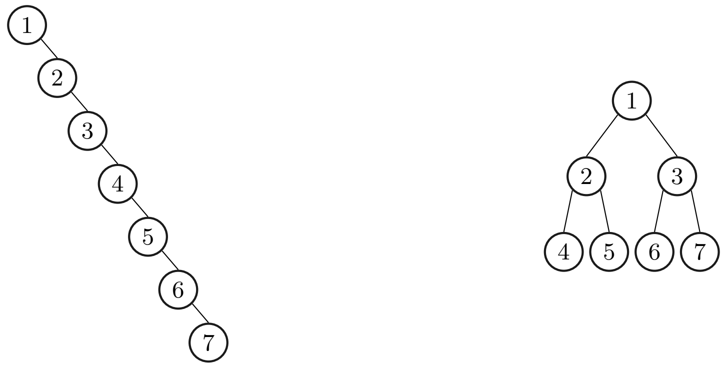

Figure 34: Two kind of trees containing numbers $1, 2, 3, 4, 5, 6, 7$.

For n numbers, the probability of any permutation of the numbers $(1, \ldots, n)$ is $\frac{1}{n!}$. Let H_k be the harmonic number defined as

$$H_k = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k}$$

For the harmonic number H_k the following equation holds:

$$\ln k < H_k \leq \ln k + 1$$

Lemma 5. In a random binary search tree (thus for a treap too) of size n, for any $k \in [0, \ldots, n-1]$ the expected height of the subtree at x is $H_{k+1} + H_{n-k} = O(1)$.

To search a key, the standard BST searching can be used.

15.1 Inserting node

Inserting a key K into a treap T goes as in a BST, by creating a leaf $x \in T$ such that $x.k = K$. At this point the binary search tree property holds, but not the heap property. So, $x.l$ can be randomly determined, then the heap property to be fixed by going up to the root and performing left or right rotations. If x is the left child of $x.p$ and $x.k < x.p.k$, then by doing RotationRight($x.p$) the heap property will be fixed and the BST property remain to hold. Conversely, if x is the right child of $x.p$ and $x.k < x.p.k$, then RotationLeft($x.p$) fixes both heap and BST properties. The insert procedure proceeds on $x.p$ until $x.k \geq x.p.k$.

15.2 Deleting node

Removing a node x from a treap T goes by moving x to T's bottom until it becomes leaf. While moving it downwards, the following situations may happen:

1. Both $x.c_l$ and $x.c_r$ are null, then no more rotations is needed and x can be deleted.
2. If $x.c_l$ (or $x.c_r$) is null, then perform the right (or left) rotation at x and proceed with these steps at x.

3. If $x.c_l.l < x.c_r.l$ (or $x.c_l.l > x.c_r.l$), then perform the right (or left rotation) at x and proceed with these steps at x.

By deleting x as leaf, both the BST and heap properties remain valid.

15.3 Splitting

To split a treap T at the given key K into two treaps T_1, T_2 such that all keys from T_1 (or T_2) are smaller (or greater) than K, insert a node x with the key K into T. The child $x.c_l$ is actually T_1 and $x.c_r$ is actually T_2.

15.4 Merging

Given two treaps T_1, T_2 where all keys from T_1 are smaller from those of T_2, they can be merged in $O(\lg n)$ time. A new node x is created such that $x.k > \max\{k \in T_1\}$ and $x.k < \min\{k \in T_2\}$. Assign the maximum priority to $x.l$ and set $x.c_l = T_1, x.c_r = T_2$. Rotations at x is made as necessary to fix the heap order. After these rotatations, x is a leaf, it can be deleted and the merging is done.

Contents

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: Introduction to Algorithms

[2] Miodrag Živković: Algoritmi

[3] Milan Vugdelija: Dinamičko programiranje

[4] Robert Sedgewick: Algorithms

[5] Daniel Dominic Sleator, Robert Endre Tarjan: Self-Adjusting Binary Search Trees

[6] Douglas Comer: The Ubiquitous B-Tree