

УНИВЕРЗИТЕТ У БЕОГРАДУ  
МАТЕМАТИЧКИ ФАКУЛТЕТ



Бојан Барџић

ИМПЛЕМЕНТАЦИЈА ТЕКСТ ЕДИТОРА ЗА  
ПИСАЊЕ КОДА

мастер рад

Београд, 2025.

**Ментор:**

др Весна МАРИНКОВИЋ, доцент  
Универзитет у Београду, Математички факултет

**Чланови комисије:**

др Милан БАНКОВИЋ, доцент  
Универзитет у Београду, Математички факултет

др Иван ЧУКИЋ, доцент  
Универзитет у Београду, Математички факултет

**Датум одбране:** 15. јануар 2016.



**Наслов мастер рада:** Имплементација текст едитора за писање кода

**Резиме:**

Кроз историју рачунарства текст едитори имају значајну улогу. Нарочито је значајан допринос текст едитора за писање кода, који представљају основни алат сваког програмера при развоју софтвера.

Циљ овог рада је упознавање са интерним функционисањем свих делова који чине један такав текст едитор. С тим у виду, у овом раду представљена је имплементација текст едитора намењеног за писање кода. Едитор је имплементиран у језику *C++* и користи неке додатне библиотеке за графички приказ. Од структура података које се користе за интерну репрезентацију текста, преко основних функционалности за измену и чување садржаја текстуалних датотека, па све до напредних функционалности које пружају удобније писање кода, у раду су описани сви детаљи овог текст едитора.

**Кључне речи:** Текст едитор, текст, имплементација, кôд, функционалност

# Садржај

<b>1</b>	<b>Увод</b>	<b>1</b>
1.1	Текст едитори . . . . .	1
<b>2</b>	<b>Структуре података у текст едиторима</b>	<b>4</b>
2.1	Бафер са размацима . . . . .	4
2.2	Уже . . . . .	5
2.3	Табела делова . . . . .	11
<b>3</b>	<b>Основне функционалности</b>	<b>16</b>
3.1	Обрада уноса . . . . .	16
3.2	Курсор . . . . .	18
3.3	Операције са појединачним карактерима . . . . .	20
3.4	Означавање текста . . . . .	22
3.5	Управљање текстуалним датотекама . . . . .	24
3.6	Исецање, копирање и налепљивање . . . . .	31
<b>4</b>	<b>Напредне функционалности</b>	<b>33</b>
4.1	Поништавање и понављање . . . . .	33
4.2	Назначавање синтаксе . . . . .	38
4.3	Правоугаоно означавање . . . . .	43
4.4	Подела екрана . . . . .	45
4.5	Сужавање простора за измену . . . . .	46
4.6	Чување исечака кода . . . . .	47
4.7	Сакривање блокова . . . . .	48
<b>5</b>	<b>Закључак</b>	<b>55</b>
	<b>Библиографија</b>	<b>57</b>

# Глава 1

## Увод

### 1.1 Текст едитори

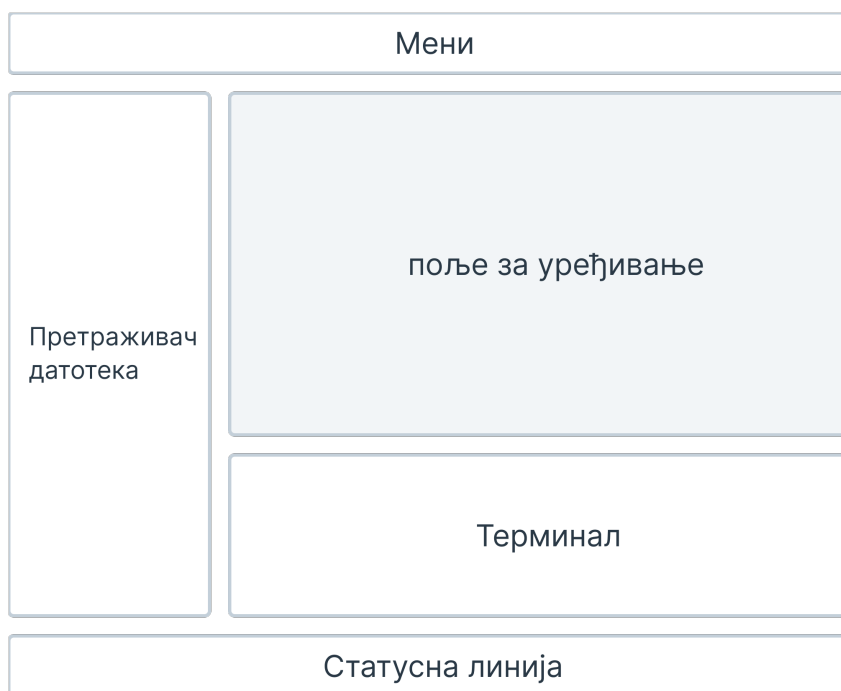
*Текст едитор* је програм за креирање, мењање, измену и прегледање текстуалних датотека. Најчешћи типови датотека који се обрађују у текст едиторима су једноставне текстуалне датотеке, датотеке које садрже изворни кôд, датотеке које садрже кôд језика за означавање као и конфигурационе датотеке. Неки од најпознатијих текст едитора су *Visual Studio Code* [1], *Notepad* [5], *Notepad++* [3], *VIM* [6] и *Emacs* [2].

Постоји више врста текст едитора. Постоје едитори *једноставног текста* (енг. *plain text*), који омогућавају уређивање датотека које садрже информације само о тексту, а не садрже информације о изгледу текста. Поред њих, постоје и едитори *богатог текста* (енг. *rich text*), где информација о датотеци поред текста садржи и неке додатне информације везано за изглед текста (фонт, величина слова, боја, маргине). Овај рад ће се бавити искључиво едиторима једноставног текста.

Ранији текст едитори били су приказивани преко стандардног излаза *терминала*. Пример једног оваквог едитора је *VIM* [6]. Тек касније се јављају едитори са *графичким корисничким окружењем* (енг. *graphical user interface*). Први такви едитори углавном су се састојали од *поља за уређивање* (енг. *editing area*), на ком је приказан садржај отворене датотеке коју корисник уређује. Поље за уређивање представља главну компоненту сваког едитора.

Поред тога, једне од основних компоненти корисничког окружења су *мени* (енг. *menu*) и *статусна линија* (енг. *status bar*). Мени се обично налази испод насловне линије и преко њега се приступа разним функционалностима које текст едитор нуди кориснику као што су манипулација датотекама, операције са привременом меморијом (енг. *clipboard*), претрага итд. Статусна линија се налази на дну текст едитора и на њој су исписане информације као што су име отворене датотеке, положај *курсора* (енг. *cursor*), енкодирање итд.

Временом се, упоредо са развојем текст едитора намењеним специфично за писање програмског кода, корисничко окружење проширује додатним компонентама као што су терминал, претраживач датотека итд. На слици 1.1 илустрован је распоред ових компоненти на примеру текст едитора за писање кода.



Слика 1.1: Распоред компоненти текст едитора за писање кода

Неке од основних функционалности којима сваки едитор треба да располаже су креирање, отварање и измена датотека, операције са привременом меморијом, претрага, поништавање и понављање. Временом су неке од на-

преднијих функционалности у текст едиторима за писање кода, као што су назначавање кода и аутоматско комплетирање текста као и функционалности везане за изглед едитора (тема, избор фонта, подешавање величине слова) постале подразумеване од стране корисника.

Када се говори о текст едиторима за писање кода, развој додатних напредних функционалности довео је до појаве великог броја различитих текст едитора. Сваки од њих је нудио како одређени скуп истих функционалности, тако и напредне функционалности специфичне за тај едитор. Такође је сваки едитор имао своје специфично корисничко окружење, потенцијално различито од осталих. Сваки од нових едитора имао је за циљ да унапреди искуство писања кода у односу на популарне текст едиторе у то време на основу својих иновација. Због различитих преференци корисника при писању кода и при избору корисничког интерфејса данас постоји разноврстан скуп популарних текст едитора.

У данашње време текст едитори имају велики значај у разним областима. Осим што су неизоставни део алата које користе програмери, својим функционалностима нуде удобније и брже писање кода, користе се у разним пословним процесима за писање докумената, у образовању, као и за писање научних радова. Изузетно је ретко пронаћи корисника који ниједном током рада на рачунару није имао потребу за текст едитором.

Корисници у данашње време често едиторе текста сматрају стандардним алатима, не удубљујући се у њихово функционисање. Наиме, едитори текста су један од основних програма који постоји на сваком функционалном рачунару. Циљ овог рада је имплементација новог едитора текста под именом *JET* (једноставан едитор текста) како би се прошло кроз имплементацију свих детаља потребних за израду једног едитора текста и стекло боље разумевање шта се дешава „испод хаубе” програма који се свакодневно употребљава од стране корисника. Едитор текста JET ће располагати и неким од нестандартних функционалности, које могу бити корисне при писању кода.



## Глава 2

# Структуре података у текст едиторима

Уколико постоји низ карактера дужине  $n$ , основне операције измене над њим су уметање неког низа карактера почев од неког индекса или брисање неког подниза карактера из њега. Пошто се текст у датотеци може посматрати као линеарни низ карактера, све измене које се врше над текстуалном датотеком се могу посматрати на овај начин. Када би комплетан садржај текстуалне датотеке која је отворена био чуван као јединствен низ карактера, операције уметања и брисања текста биле би временски јако скупе (реда  $O(n)$ , где је  $n$  дужина текста) и њихово узастопно извршавање над неким дугачким текстом би имало за последицу изузетну неефикасност едитора текста. Како би се овај проблем превазишао осмишљене су различите структуре података које ове операције врше ефикасније. Најпознатије међу њима су *бафер са размацима* (енг. *gap buffer*), *уже* (енг. *rope*) и *табела делова* (енг. *piece table*).

### 2.1 Бафер са размацима

Бафер са размацима представља одређени вид проширења низа карактера који омогућава ефикасније операције уметања и брисања у ситуацији када се измене врше на блиским локацијама у тексту. Ова структура података се заснива на идеји да постоји један линеаран низ чија је средина „празна”, док се са леве и десне стране налази текст. Како се врше измене над неким делом текста тако се средина „помера” превлачењем последњег елемента леве стране на почетак десне или обрнуто. Када се бафер попуни (размак није довољно

велики за нову операцију), тада се алоцира нови бафер већих димензија, најчешће двоструко веће димензије, и у њега се копира текст из старог бафера.

У односу на класичан низ карактера, операције над бафером са размацима су временски доста ефикасније јер се не захтева реалокација низа при свакој измени. Ефикасност операција уметања и брисања пре свега зависи од тога колики је размак између индекса на коме се врши измена текста и леве или десне границе размака. Ако је размак већ на потребној позицији, временска сложеност ове операције је  $O(1)$ . Међутим, ако је размак на једном крају а потребно је да се мења други крај текста сложеност ће бити  $O(n)$ . У просечном случају временска сложеност наведених операција је константна, јер се карактери најчешће пишу (и бришу) један за другим. Треба напоменути и то да је с времена на време потребно вршити реалокацију низа, која је линеарне сложености у односу на дужину текста.

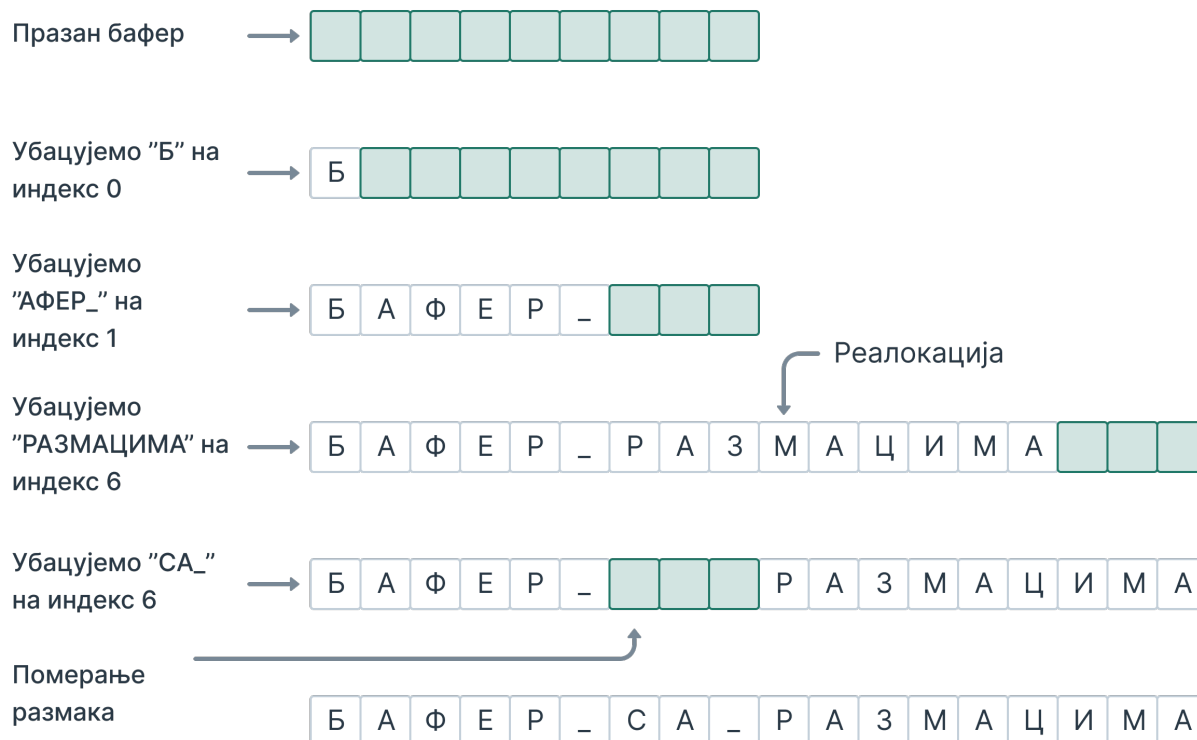
Разматрана структура података је једноставна за имплементацију и чест је избор при имплементацији текст едитора. Познати текст едитори *Emacs* [2] и *Notepad++* [3] користе ову структуру података у својој имплементацији.

**Пример 1.** На слици 2.1 приказана је илустрација рада бафера са размаком. У првом кораку бафер је празан. У следећем кораку се додаје ниска „б” на почетну позицију. После тога на крај текста се додаје ниска „афер\_”, а после тога и ниска „размацима”. Пошто је бафер у том тренутку пун, врши се реалокација. Када се текст пребаци у нови низ, помера се почетак празног дела на жељени индекс и врши се уметање нове ниске. У последњем кораку се додаје ниска „са\_” на индекс број 6.

## 2.2 Уже

Уже је бинарно стабло у коме сви чворови који нису листови садрже број карактера у левом подстаблу тог чвора. У листовима се налазе ниске које садрже делове текста. Када се прође кроз листове редом од првог листа слева до крајњег десног листа, добија се целокупан текст.

Предност ове структуре података у односу на обичан низ карактера је што операције као што су уметање, брисање и претрага текста у просечном случају



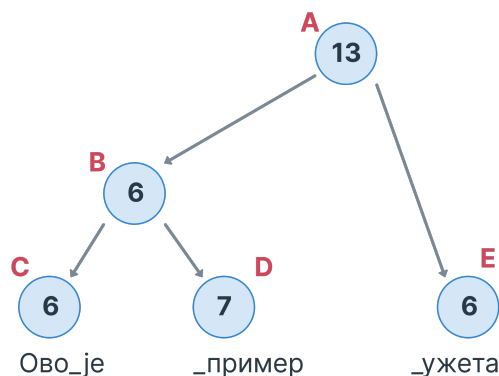
Слика 2.1: Илустрација бафера са размаком

имају временску сложеност  $O(\log n)$  (где је  $n$  дужина текста), а не  $O(n)$ . Такође, није потребно чувати текст у непрекидном делу меморије, већ делови текста могу бити разбацани по меморији.

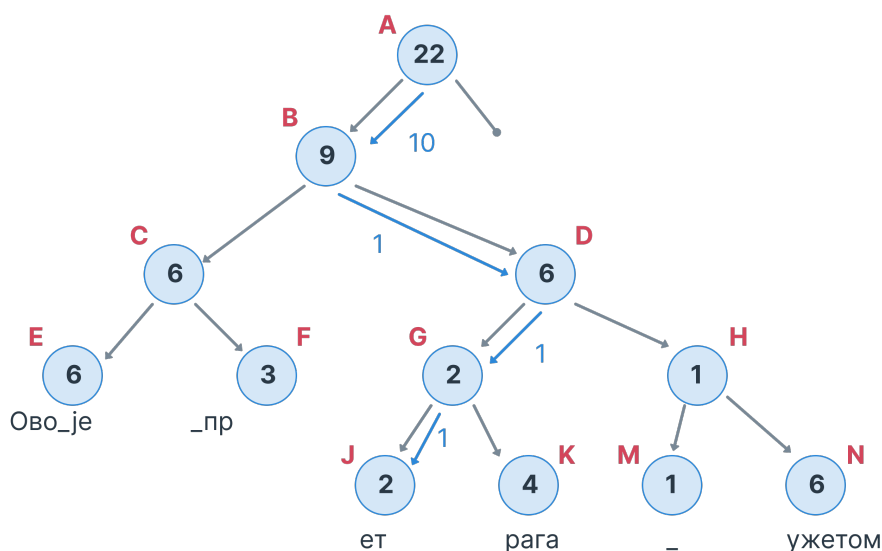
Лоше стране ужета као структуре података су нешто сложенија структура, због чега су чешће грешке како у раду са њом, тако и при њеној имплементацији. Поред тога, заузима више меморије него обичан низ карактера (због потребе за чувањем унутрашњих чворова стабла). Пример текст едитора који користи уже у својој имплементацији је  $X_i$  [4]. На слици 2.2 приказан је пример ужета који садржи текст „Ово је пример ужетѡа”.

## Претрага

Приступ карактеру на  $i$ -том индексу се врши тако што се рекурзивно пролази кроз стабло почевши од корена. Ако је индекс мањи од вредности у



Слика 2.2: Пример једног ужета



Слика 2.3: Претрага помоћу ужета

текућем чвору, настављамо претрагу у његовом левом подстаблу. У супротном од  $i$  се одузима вредност коју чува тренутни чвор и наставља се претрага у његовом десном подстаблу. Када се наиђе на лист враћа се  $i$ -ти карактер из ниске која се налази у чвору. Сложеност операције је иста као код претраге бинарног стабла, а то је  $O(\log n)$  за балансирано стабло.

**Пример 2.** Пример претраге ужета илустриран је на слици 2.3. Нека је из ужета које садржи текст „Ово је претрага ужетом” пошребно дохваћени карактер текста на индексу 10, тј. карактер „ш”. Претрага креће из корена који садржи дужину укупног текста, па пошто је тражени индекс мањи

од вредности у корену, прелазе се наставља у његовом левом подстаблу. Следећи чвор садржи вредност 9, што значи да је изражени индекс већи од вредности. Од вредности 10 израженог индекса се одузима вредност у текућем чвору и прелазе се наставља у десном подстаблу тренутног чвора. Сада је изражени индекс 1 и он се пореди са вредношћу у тренутном чвору. Пошто је 1 мање од 6 прелази се у лево подстабло. У следећем чвору вредност је 2 па се поново иде у лево подстабло. На крају се налази на листи који садржи ниску „eи” и дохвата се карактер на индексу 1 тј. карактер „и”, што је исправна вредност.

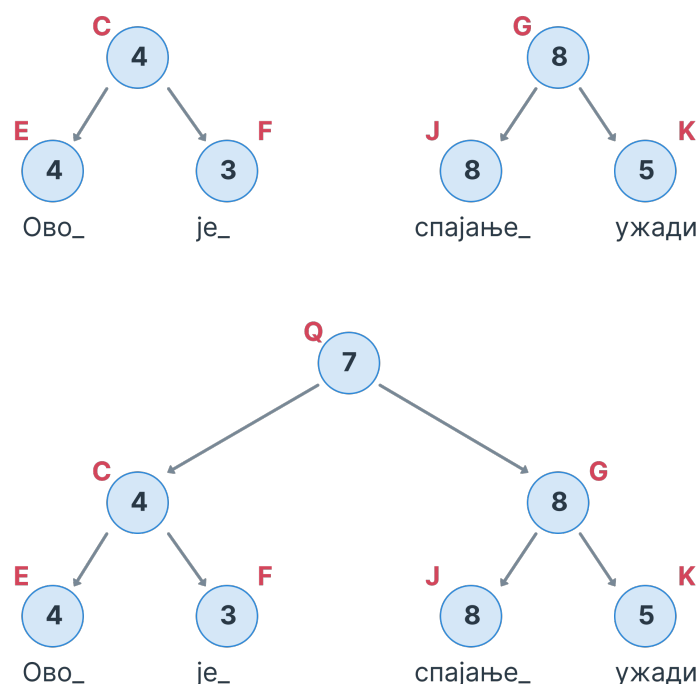
Пре него што буде обрађена операција брисања, прво ће бити уведене две нове операције које ће бити потребне за њену реализацију. Прва операција је надовезивање једног ужета на друго, док је друга операција дељење једног ужета на два нова ужета по неком карактеру.

## Надовезивање

Надовезивање (енг. *concatenation*) је операција којом се на уже, које садржи текст  $t_1$ , надовезује друго уже које садржи текст  $t_2$  и добија се ново уже које садржи текст  $t_1t_2$ . Надовезивање два ужета,  $r_1$  и  $r_2$ , се врши тако што се направи нови родитељски чвор чије ће лево дете бити корен од  $r_1$ , а десно корен од  $r_2$ . Вредност новог чвора ће бити сума дужина ниски у свим листовима ужета  $r_1$ . Да би се израчунала вредност новог корена потребно је израчунати суму дужина ниски свих листова који се налазе у  $r_1$ . То се постиже рекурзивним обиласком  $r_1$ , где се на коначну суму додаје вредност тренутног чвора и затим се рекурзивно обилази десно подстабло тренутног чвора. Сложеност ове операције је  $O(\log n)$  за балансирано стабло. Пример надовезивања два ужета се може видети на слици 2.4

## Дељење

Дељење (енг. *splitting*) ужета  $r$  који садржи текст  $t$  по индексу  $i$  на два ужета,  $r_1$  и  $r_2$ , који садрже текстове  $t_1$  и  $t_2$ , где  $t_1$  садржи карактере од почетка текста  $t$  до  $i$ -тог индекса (укључујући и карактер на  $i$ -том индексу), а  $t_2$  садржи карактере десно од  $i$ -тог индекса па до краја текста  $t$ , врши се на следећи начин. Најпре се налази  $i$ -ти карактер: ако је то последњи карактер



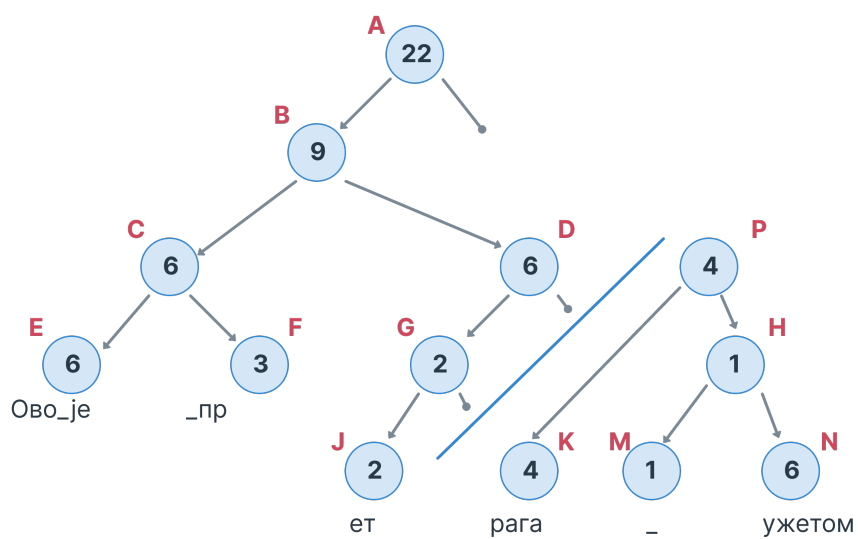
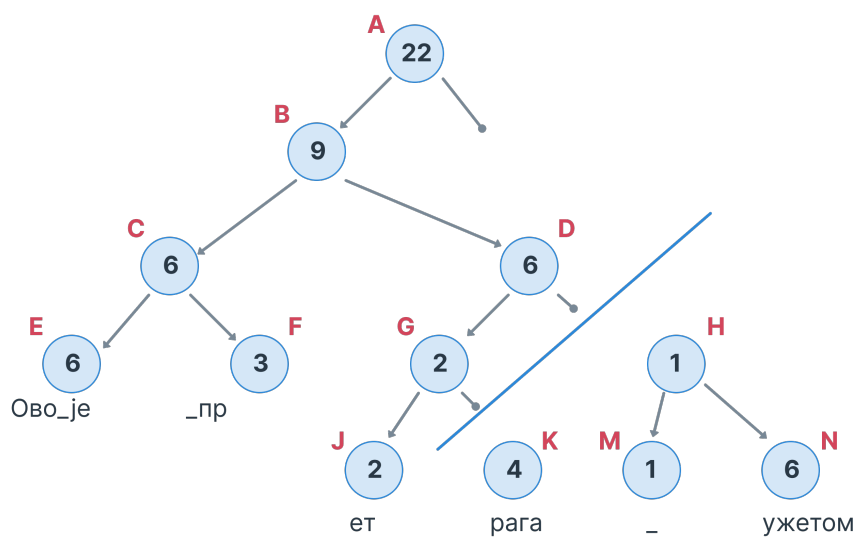
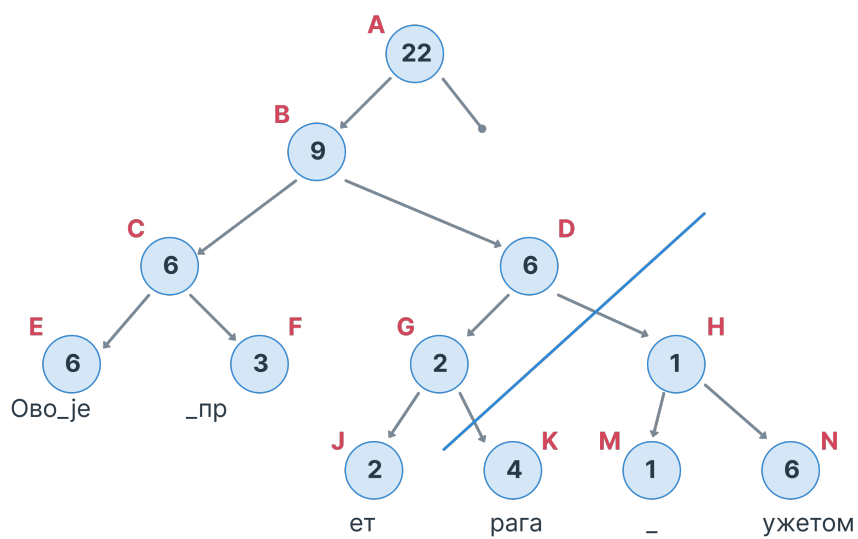
Слика 2.4: Надовезивање два ужета

ниске у листу онда се не ради ништа. У супротном се лист дели на два нова листа, где је  $i$ -ти карактер последњи карактер ниске левог листа. Нека је лист чији је  $i$ -ти карактер последњи у нисци означен са  $l_s$ . Листови стабла деле се у две групе  $L_1$  и  $L_2$ , тако да се у  $L_1$  налазе листови лево од  $l_s$  као и сам  $l_s$ , док се у  $L_2$  налазе листови десно од  $l_s$ . Листови из групе  $L_2$  се одвајају од главног ужета и затим се међусобно спајају. Алгоритам се завршава тако што се балансирају два новодобијена ужета. Сложеност операције дељења ужета је иста као и код претходно разматраних операција.

**Пример 3.** На слици 2.5 је илустриран начин дељења по индексу 10. Прво се проналази карактер на датом индексу на начин описан у делу о преирази. Затим се проверава да ли је тај карактер последњи у листу, па пошто јесте, врши се подела на  $l_1$  и  $l_2$  по том листу.

## Уметање

Да би се неки текст  $t$  уметнуо у уже  $r$  почев од индекса  $i$ , довољно је да се искористе претходно дефинисане операције дељења и конкатенације. Најпре



Слика 2.5: Дељење ужета по индексу 10

се уже  $r$  подели по индексу  $i$ , чиме се добијају два ужета  $r_1$  и  $r_2$ , затим се на  $r_1$  надовеже текст  $t$  чиме се добија ново уже  $r_3$ . На крају се на уже  $r_3$  надовеже  $r_2$ . Ова операција се састоји од три операције сложености  $O(\log n)$ , тако да је њена укупна сложеност  $O(\log n)$ .

### Брисање

Уколико је потребно обрисати сегмент текста  $t$  смештеног у ужету  $r$  који почиње на  $i$ -том карактеру, а завршава се на  $(i + l)$ -том карактеру, онда се то може урадити у три корака. Прво се врши дељење ужета  $r$  по индексу  $i$  на два ужета  $r_1$  и  $r_2$ , затим се уже  $r_2$  дели по  $l$ -том индексу на  $r_3$  и  $r_4$ . У последњем кораку, надовезивањем  $r_1$  и  $r_4$  се добија коначан резултат. Из истих разлога као код уметања, сложеност операције брисања је  $O(\log n)$ .

## 2.3 Табела делова

*Табела делова* (енг. *piece table*) је структура података која се састоји од два бафера у којима се налази текст и повезане листе чворова који показују на текст у баферу. Текст едитор који је имплементиран током рада на овој тези користи ову структуру података, тако да ће она бити нешто детаљније описана него претходне две структуре података.

У први бафер се учитава текст који се већ налазио у датотеци коју смо отворили и тај бафер се назива *ориџинални бафер* (енг. *original buffer*). Од тренутка учитавања текста из датотеке па надаље он остаје непромењен.

У други бафер, који се назива *бафер за додавање* (енг. *add buffer*), додаје се текст који се током времена уписује у текст едитор. Сваки пут када се додаје текст, без обзира на то на ком месту теба извршити додавање, текст се додаје на крај бафера. Важно је напоменути да када се брише текст, он се не брише из бафера за додавање, већ остаје у њему. На овај начин се ни у ком моменту не јавља потреба да се врши померање текста који се већ налази унутар бафера.

Поставља се питање како је онда могуће исписати текст у правилном редоследу. То се постиже коришћењем двоструко повезане листе чији су елементи



тзв. *дескриптори делова* (енг. *piece descriptors*). Сваки дескриптор садржи информацију о томе на који од два бафера показује, на којем индексу бафера почиње текст и колика је дужина тог текста. Следећи код приказује чланске променљиве класе `PieceDescriptor` у имплементацији рада.

```
class PieceDescriptor {  
    // ...  
private:  
    SourceType m_source;  
    size_t m_start;  
    size_t m_length;  
};
```

Предност ове структуре је што је једноставним операцијама, уметањем и брисањем чворова из повезане листе, могуће ефикасно вршити измене текста. Мана је потенцијално велико меморијско заузеће бафера за додавање као и фрагментација на доста веома малих „делова“, што чини претрагу кроз листу мање ефикасном.

У наставку текста биће описане основне операције над овом структуром података. Најпре ће бити уведене неке ознаке које ће бити употребљаване за све наведене операције:

- $n$  - дужина укупног текста,
- $m$  - број елемената повезане листе,
- $I_n \in \{0, 1, \dots, n - 1\}$  - скуп валидних индекса текста,
- $I_m \in \{0, 1, \dots, m - 1\}$  - скуп валидних индекса повезане листе,
- $p_i$  - почетак  $i$ -тог дескриптора, где је  $i \in I_m$ ,
- $d_i$  - дужина  $i$ -тог дескриптора, где је  $i \in I_m$ .

### Уношење текста

Нека је потребно додати неки текст дужине  $d$  у текући текст почев од позиције  $i \in \{0, 1, \dots, n\}$ . Размотримо најпре два специјалана случаја за индекс  $i$ :

1.  $i = 0$ : У овом случају се додаје нови дескриптор на почетак листе.
2.  $i = n$ : У овом случају се додаје нови дескриптор на крај листе.

За све остале случајеве пролази се кроз листу дескриптора слева на десно. Нека је сума дужина свих дескриптора пре  $j$ -тог, при чему је  $j \in I_m$ , једнака  $s_j$ . Зауостављамо се када буде важио услов  $s_j + d_j \geq i$ , где је  $j$  индекс елемента листе на ком се налазимо. Разликујемо два случаја:

1.  $s_j + d_j = i$ : У овом случају убацује се нови дескриптор испред  $j$ -тог елемента повезане листе дескриптора.
2.  $s_j + d_j > i$ : У овом случају се тренутни дескриптор дели на два тако да леви садржи  $i - s_j$  почетних карактера оригиналног дескриптора, а десни остале. Ово се постиже тако што се дужина  $j$ -тог дескриптора поставља на  $i - s_j$ , затим се прави нови дескриптор чији је почетак  $p_j + (i - s_j)$ , а дужина  $d_j - (i - s_j)$  и умеће се испред  $j$ -тог. На крају се додаје дескриптор са новим текстом између ова два.

Сложеност ове операције је  $O(m + d)$ , јер се кроз листу пролази највише  $m$  пута. Додавање новог дескриптора и дељење претходног на два су операције сложености  $O(1)$ , док додавање новог текста дужине  $d$  у бафер има сложеност  $O(d)$ .

## Брисање

Брисање дела текста чији се индекси налазе у целобројном интервалу  $[p, k)$ , где су  $p \in I_n$ ,  $k \in I_n \cup \{n\}$ , се врши аналогно уметању текста. Пролази се кроз повезану листу све док не буде испуњен услов  $s_i + d_i \geq p$  за неко  $i \in I_m$ . Затим се редом пролази кроз све дескрипторе који садрже текст чији су индекси из опсега  $[p_j, p_j + d_j)$  за  $j \geq i$ ,  $j \in I_m$  и имају пресек са  $[p, k)$  и врши се њихово ажурирање на основу следећих правила:

1. Ако је пресек суфикс опсега  $[p_j, p_j + d_j)$  дужине  $d_p$ , онда се одузима суфикс од дескриптора тако што му се смањује дужина за  $d_p$ .
2. Ако је пресек префикс опсега  $[p_j, p_j + d_j)$  дужине  $d_p$ , онда се одузима префикс од дескриптора тако што му се помера почетак у десно за  $d_p$  и смањује дужина за исто толико.

3. Ако је пресек цео опсег  $[p_j, p_j + d_j)$ , онда се брише цео дескриптор.
4. Ако је  $[p, k)$  садржан у  $[p_j, p_j + d_j)$  и није ни префикс ни суфикс опсега, онда се дескриптор дели на два нова, где први садржи опсег  $[p_j, p)$ , а други  $[k, p_j + d_j)$ .

Пошто се брисањем мења један или више дескриптора у повезаној листи, треба приметити нека правила везана за горе наведене случајеве. Ако се мења више дескриптора у једном брисању, онда се они морају налазити на узастопним позицијама у повезаној листи. Случај 1. може важити само за последњи дескриптор који ће бити промењен у брисању. Аналогно, случај 2. може важити само за први који ће бити промењен, док случај 3. може важити за све које ће се променити. Уколико за неки дескриптор важи случај 4, онда је он једини дескриптор који ће се ажурирати.

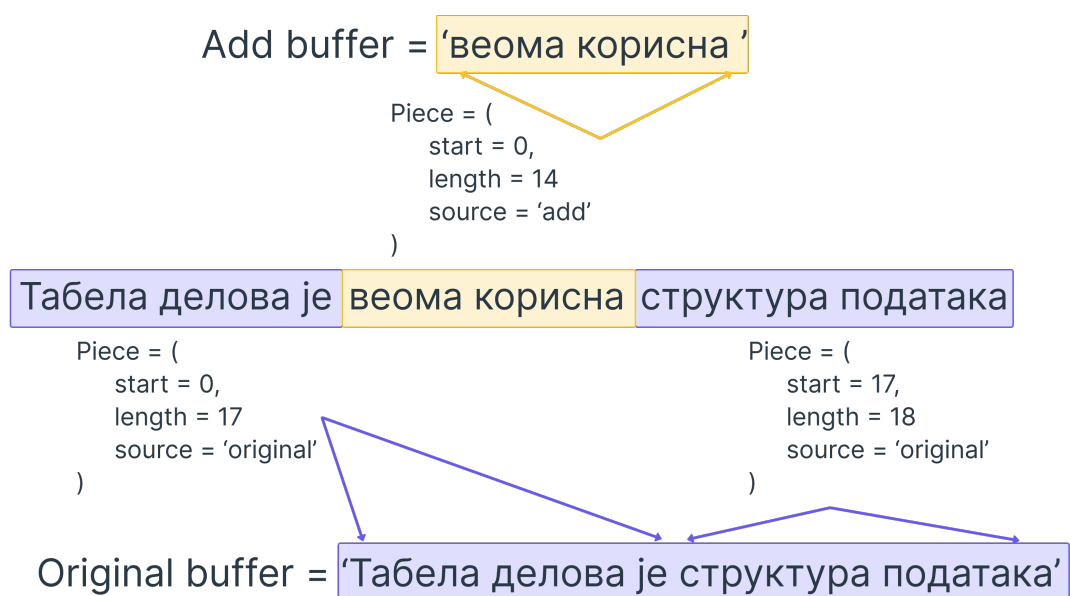
Временска сложеност брисања текста је  $O(m)$ , јер се пролази кроз највише  $m$  елемената повезане листе и свако ажурирање дескриптора је сложености  $O(1)$ .

## Исписивање

Уколико је потребно да се испише целокупан текст на стандардни излаз или у неку датотеку, то је могуће постићи на следећи начин. Пролази се кроз листу дескриптора од почетка до краја и за сваки дескриптор се исписује подниска одговарајућег бафера која почиње на индексу  $p_j$  и има дужину  $d_j$ , где је  $j \in I_m$ . На слици 2.6 се може видети како изгледа приказ једног текста помоћу табеле делова.

По овом механизму по коме се целокупан текст добија надовезивањем „делова” бафера је сама структура података добила име. Један од познатијих текст едитора који од 2018. користи табелу делова у својој имплементацији је *Visual Studio Code* [1].

Временска сложеност операције исписивања садржаја табеле делова је  $O(\sum_{i=0}^{m-1} d_i)$  тј. једнака је суми дужина свих дескриптора. Пошто је сума свих дужина једнака дужини укупног текста, онда се сложеност може једноставније записати као  $O(n)$ .



Слика 2.6: Приказ рада табеле делова

## Глава 3

# Основне функционалности

### 3.1 Обрада уноса

Током уноса и обраде текста коришћењем едитора текста, корисник већину команди задаје преко тастатуре, док се мањи проценат њих задаје коришћењем миша. Један од задатака при имплементацији текст едитора јесте исправно обрадити све овакве уносе. Пре описа обраде, прво ће бити речи о самим карактерима и њиховим специфичностима.

#### Карактери

За писање програмског кода најчешће се користи *ASCII* кодирање. *ASCII* табела представља стандард за кодирање знакова који се користи за представљање текста у рачунарима. Она пресликава знакове (слова, цифре, знакове интерпункције, контролне секвенце, итд.) у бројевне вредности, омогућавајући на тај начин њихово чување и манипулацију у дигиталним системима. *ASCII* стандард користи 7 битова за представљање сваког карактера, омогућавајући представљање 128 различитих карактера.

Немају сви карактери из *ASCII* табеле визуелну репрезентацију: карактери чија је вредност у табели од 0 до 31 су тзв. *контролни карактери* (енг. *control characters*) који служе за слање сигнала за комуникацију са периферним уређајима. Остали карактери, чија је вредност између 32 и 127, се могу приказивати на излазу и њих називамо *штампањим карактерима* (енг. *printable characters*). У табели 3.1 приказан је садржај *ASCII* табеле.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht	lf	vt	ff	cr	so	si
1x	dle	dc1	dc2	dc3	dc4	nak	syn	etb	can	em	sub	esc	fs	gs	rs	us
2x	sp	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	del

Табела 3.1: *ASCII* табела.

## Унос помоћу тастатуре

Уколико корисник притисне неки тастер на тастатури, текст едитор ће га исписати само ако тастер одговара штампаном карактеру или карактеру за форматирање текста (као што су *Tab* и *Enter*). Сви остали тастери се игноришу што омогућава да се неким од њих кодирају друге команде. Поред тога, могуће је комбиновање контролног и штампаног карактера како би се кодирале додатне команде. Ово се постиже притиском на два или три тастера истовремено. Најчешће се врши комбиновање два тастера, где је први обично *Ctrl* (нпр. *Ctrl+N*, *Ctrl+Z*, *Ctrl+C*).

У едитору текста имплементираног при раду на тези, обраду ових уноса врши корисничка метода `handleKeyboardInput()` класе `TextEditor`. Она врши периодичну проверу да ли је притиснут један или више тастера у датом тренутку и да ли је том комбинацијом тастера задата нека специјална команда. Уколико јесте, извршава се одговарајућа команда. Ако то није случај, утврђује се да ли је притиснут неки штампани карактер и уколико јесте, прослеђује се методи за унос.

### Унос помоћу миша

Некада се команде задају и путем миша, најчешће ради померања *курсора* (енг. *cursor*) и *означавања текста* (енг. *text selection*). Помоћу миша се такође може приступити менију у коме се налазе додатне команде и алати. О курсору и означавању, као и додатним командама биће више речи у наставку рада. Сигнали који се обрађују су тастер миша који је притиснут (леви, десни или средњи), позиција курсора миша, као и то да ли корисник користи један клик, дупли клик или држи тастер миша.

Имплементација обраде уноса миша је сложенија него код обраде уноса са тастатуре. Метода `handleMouseEvent()` класе `TextEditor` обрађује све догађаје миша. Она се периодично позива и врши проверу да ли је корисник притиснуо леви тастер на мишу и уколико јесте, активира се одговарајућа команда и прослеђује се позиција миша. Такође се проверава да ли је корисник држао леви тастер и уколико јесте, активира се одговарајућа метода и шаљу две позиције миша: у тренутку када је дугме притиснуто и када је отпуштено. Још једна врста обраде уноса миша је померање *точкића миша* (енг. *mouse wheel*) којим се врши хоризонтално и вертикално скривање (енг. *scrolling*)<sup>1</sup>.

## 3.2 Курсор

Један од основних елемената сваког едитора текста јесте курсор. Његова функција је да означаи место у тексту на којем ће бити извршена следећа измена. Углавном је представљен у виду уређеног пара реда и колоне на којима се курсор налази. Иако су у овом раду до сада места на којима се врше измене текста референцирана помоћу индекса јединственог низа карактера, репрезентација у виду уређеног пара је интуитивнија и прегледнија за корисника. Стога се ова репрезентација користи и у класи `TextCoordinates` едитора текста имплементационог при раду на тези.

```
class TextCoordinates {  
    // ...
```

---

<sup>1</sup>Притисак на десни тастер миша није обрађен у имплементацији развијеног текст едитора.

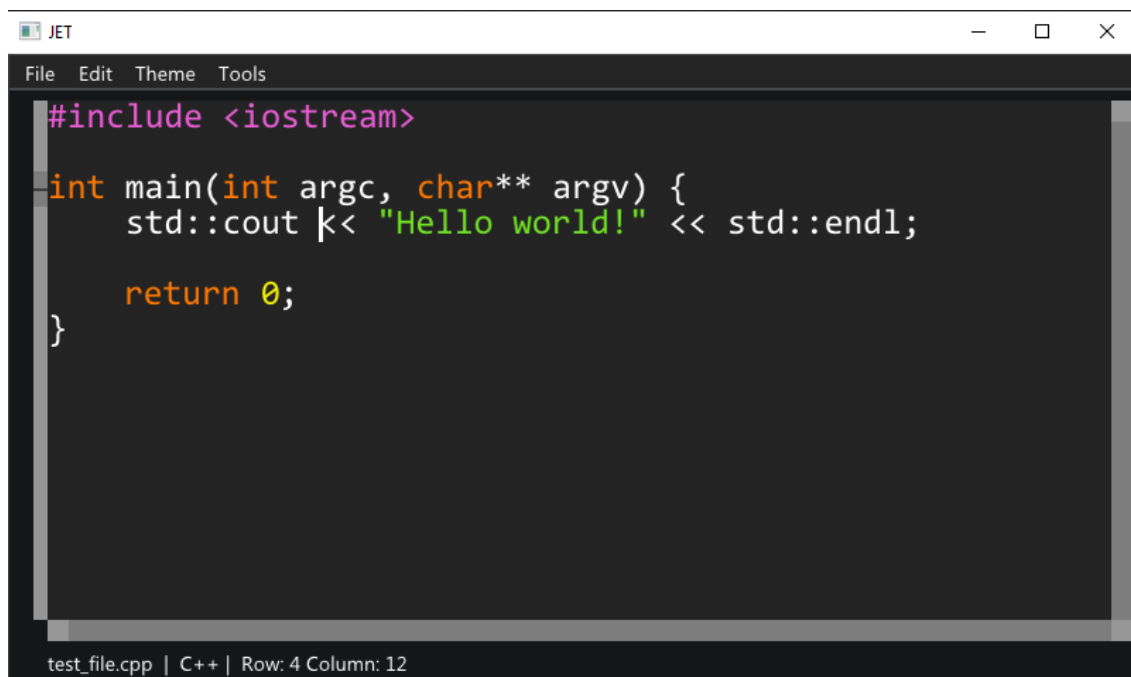
```
public:
    size_t m_row;
    size_t m_col;
};
```

Треба, такође, напоменути да у овој репрезентацији обе координате почињу од један, а не од нула као код индекса, као и да се вредност колоне може кретати у целобројном интервалу  $[1, d + 1]$ , где је  $d$  дужина тренутног реда на ком се курсор налази.

Курсор се на већини данашњих рачунара помера позиционирањем миша на жељено место и притиском на леви тастер или, алтернативно, помоћу стрелица на тастатури. Такође, притиском на тастере *Home* и *End* врши се померање курсора на почетак, односно на крај тренутног реда. У наредном програмском коду илустроване су неке од метода за померање курсора, док се на слици 3.1 може видети курсор у текст едитору представљен белом усправном цртом. У статусној линији могу се видети тренутне координате курсора, тј. редни број његове врсте и колоне.

```
class Cursor {
public:
    //...
    void moveRight(size_t times = 1);
    void moveLeft(size_t times = 1);
    void moveUp(size_t times = 1);
    void moveDown(size_t times = 1);
    void moveToBeginning();
    void moveToEnd();
    void moveToEndOfFile();
    //...
};
```





Слика 3.1: Приказ курсора у текст едитору

### 3.3 Операције са појединачним карактерима

#### Унос

Једна од основних функционалности текст едитора је уношење текста карактер по карактер. Тај унос се врши тако што корисник притиска тастер за одговарајући штампани карактер и он се уноси на тренутну позицију курсора, тј. испред карактера на ком се налази курсор, уколико није на крају реда, када се уноси иза последњег карактера.

Унос се имплементира тако што се дохвате координате курсора и проследе функцији `textCoordinatesToBufferIndex(TextCoordinates coords)` која преводи уређени пар врсте и колоне у индекс континуалног низа карактера. То се постиже на следећи начин: вредност индекса се иницијализује на 0, а затим се пролази кроз све редове изнад тренутног, почевши од првог, и на индекс се додаје  $d + 1$  (додаје се један да би се урачунао и знак за нови ред), где је  $d$  дужина тог реда. Када се стигне до тренутног реда вредност индекса се увећава за вредност колоне умањене за један. Наредни програмски код

илуструје рад ове функције.

```
size_t LineBuffer::textCoordinatesToBufferIndex
(const TextCoordinates &coords) const {

    size_t index = 0;

    for (size_t i=0; i < coords.m_row-1; i++)
        index += lineAt(i).size() + 1;

    index += (coords.m_col - 1);

    return index;
}
```

Када је добијен индекс низа карактера, он се прослеђује табели делова заједно са притиснутим карактером и одговарајући карактер се уноси на одговарајућој позицији и курсор се помера за једно место у десно. На примеру функције `insertCharToPieceTable` која припада класи `TextBox` се може видети поступак уметања карактера у табелу делова.

```
bool TextBox::insertCharToPieceTable(char c) {
    auto coords = m_cursor->getCoords();
    size_t index = m_lineBuffer->textCoordinatesToBufferIndex(coords);
    return m_pieceTableInstance->getInstance().insertChar(c, index);
}
```

## Брисање

Брисање појединачних карактера се врши тако што корисник позиционира курсор на жељено место и притисне тастер за брисање карактера. Треба напоменути да постоје два тастера за брисање карактера, то су *Backspace* и *Delete*. *Backspace* брише карактер лево од курсора, док *Delete* брише онај на позицији курсора. После брисања се у првом случају курсор помера за једно место у лево, док у другом случају остаје на истој позицији. Брисање је имплементирано на сличан начин као унос.

Текст едитор који је имплементиран током рада на овој тези користи две одвојене методе за обе врсте брисања. То су `backspace()` и `deleteChar()` класе `TextBox`. У наредном примеру кода приказана је имплементација функције `backspace()`.

```
void TextBox::backspace() {
    auto deleted = deleteSelection();

    if (deleted)
        return;

    auto coords = m_cursor->getCoords();
    size_t index = m_lineBuffer->textCoordinatesToBufferIndex(coords);

    if (index != 0) {
        updateUndoRedo();
        m_pieceTableInstance->getInstance().flushInsertBuffer();

        auto initialized =
            m_pieceTableInstance->getInstance().backspace(index);

        if (initialized)
            m_cursor->recordCursorPosition();

        updateStateForTextChange(false, 1, m_cursor->getRow());
        m_cursor->moveLeft();
        updateStateForCursorMovement();
    }
}
```

### 3.4 Означавање текста

Означавање текста је функционалност која кориснику пружа могућност означавања неког непрекидног сегмента текста ради брисања или *копирања* (енг. *copy*). Копирање ће бити објашњено у наставку рада. Означавање текста се најчешће врши помоћу миша, тако што корисник превлачи курсор миша преко дела екрана у коме се налази текст који жели да изабере и унесе

команду за брисање или копирање. Означавање текста се може вршити и помоћу тастатуре, тако што корисник држи тастер *Shift* и стрелицама помера курсор. У овом случају се памти позиција курсора у тренутку када је притиснут тастер и она ће представљати један крај означеног текста. Други крај означеног текста ће бити позиција на коју је корисник померио курсор држећи тастер *Shift*.

Пре него што буде објашњена имплементација, прво ће бити дефинисан поредак између текстуалних координати, о којима је било речи у одељку 3.2. Пошто су координате представљене уређеним паром реда и колоне, оне се тако и пореде. Прво се пореде вредности редова, па уколико су оне једнаке, онда се пореде вредности колоне. У следећем коду илустрована је дефиниција поређења за операторе `=` и `<`. Симетрични оператори се дефинишу аналогно.

```
bool TextCoordinates::operator==(const TextCoordinates &other) const {
    return m_row == other.m_row && m_col == other.m_col;
}

bool TextCoordinates::operator<(const TextCoordinates &other) const {
    return (m_row < other.m_row)
        || (m_row == other.m_row && m_col < other.m_col);
}
```

Означавање непрекидног дела текста је у едитору ЈЕТ представљено помоћу две текстуалне координате. Прва координата представља локацију на којој почиње текст који је означен, а друга локацију где се завршава. Треба нагласити да почетак мора бити мањи или једнак од краја. Потребно је, такође, чувати информацију о томе да ли је означени текст активан. У следећем коду илустрован је пример чланских променљивих класе `Selection`.

```
class Selection {
//...
private:
    bool m_active;
    bool m_rectangular;
    TextPosition m_start;
    TextPosition m_end;
};
```

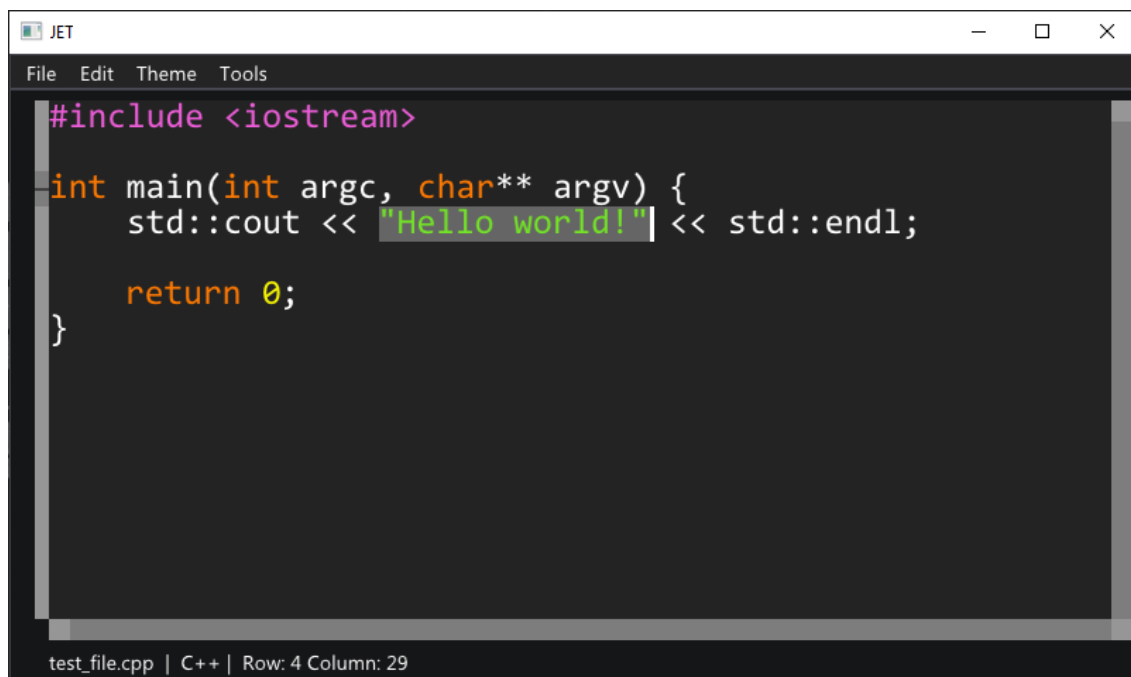
Када метода `handleMouseInput()` детектује држање левог тастера, она прослеђује две позиције миша методи `setMouseSelection()` класе `TextBox`. Онда она на основу те две позиције рачуна на које текстуалне координате оне показују и додељује мању од вредности координати која означава почетак, а већу координати која означава крај. На крају се означени текст обележава као активан.

Уколико је означени текст активан и корисник кликне било где на текстуално поље едитора, он се деактивира. Тренутно означени текст се може деактивирати и притиском на стрелице за померање курсора. При притиску на тастере за брисање док је означени текст активан, сав означени текст се брише. Исто важи и за операције *исецања* (енг. *cut*) и *налепљивања* (енг. *paste*). Деактивирање означеног текста се имплементира тако што се чланска променљива `m_active` постави на вредност `false`.

Притиском на комбинацију тастера *Shift* и *Home* се почетак означавања поставља на почетак текућег реда, а крај на тренутну позицију курсора, док се притиском на тастере *Shift* и *End* почетак означеног текста поставља на позицију курсора, а крај на последњу колону у текућем реду. Притиском на комбинацију тастера *Ctrl* и *A* означава се читав текст. На слици 3.2 се може видети пример означавања, где је означени текст уоквирен правоугаоником другачије боје од позадине.

### 3.5 Управљање текстуалним датотекама

Сврха едитора текста јесте мењање текстуалних датотека или креирање нових. Како би то било могуће, потребно је кориснику обезбедити могућност да креира нову или отвори постојећу датотеку. Потребно је и омогућити чување измена које корисник прави као и могућност избора локације на којој ће датотека бити сачувана. У наредним одељцима биће обрађене те функционалности. Овим функционалностима се приступа путем менија који се налази испод насловне линије. На слици 3.3 илустрован је мени за управљање датотекама у оквиру едитора JET.



Слика 3.2: Приказ означавања у текст едитору



Слика 3.3: Приказ менија за управљање датотекама

## Чување датотека

Корисник може сачувати тренутни садржај едитора у датотеци притиском на дугме *Save* у менију или комбинацијом тастера *Ctrl* и *S*. Тренутни садржај се чува на жељеној локацији, а уколико иста није дефинисана, тражи се од корисника да је унесе. Такође је дозвољено и чување тренутне датотеке на новој локацији притиском на дугме *Save as* у менију, при чему је поступак исти као и у случају када путања није унапред дефинисана.

Чување датотека у оквиру едитора JET извршава метода `save()` класе `TextEditor`. Она проверава да ли је дефинисана путања и уколико није, прослеђује извршавање функцији `saveAs()`. Ако је путања дефинисана, извршавање се наставља у истоименој функцији класе `TextBox`. Имплементација ове две истоимене функције, као и функције `saveToFile()`, илустрована је у следећем програмском коду.

```
void TextEditor::save() {
    if (m_activeTextBox->getFile() == nullptr)
        saveAs();
    else {
        m_activeTextBox->save();
        m_inactiveTextBox->save();
    }
}

bool TextBox::save() {
    m_dirty = false;

    m_pieceTableInstance->getInstance().flushInsertBuffer();
    m_pieceTableInstance->getInstance().flushDeleteBuffer();

    return saveToFile();
}

bool TextBox::saveToFile() {
    std::stringstream strStream;
    strStream << m_pieceTableInstance->getInstance();
    std::string buffer = strStream.str();
}
```

```

        return File::writeToFile(buffer, m_file->getPath());
    }

```

Када путања није дефинисана позива се метода `saveAs()`. Она отвара дијалог у коме корисник треба да изабере локацију на којој ће бити сачувана датотека и враћа њену путању. Уколико није дефинисана, *екстензија* (енг. *extension*) датотеке биће постављена на обичну текстуалну датотеку (*.txt*). Затим се позива истоимена метода класе `TextBox` која поставља нову путању и уписује садржај едитора на дату локацију. Имплементација метода `saveAs()` класа `TextEditor` и `TextBox` илустрована је у наредном примеру кода.

```

void TextEditor::saveAs() {
    auto path = saveFileDialog();

    if (!path.empty()) {
        if (path.find_last_of('.') == std::string::npos)
            path += ".txt";
        m_activeTextBox->saveAs(path);
        m_inactiveTextBox->saveAs(path);
    }
}

bool TextBox::saveAs(std::string &filePath) {
    auto oldFile = m_file;
    m_file = new File(filePath);
    m_lineBuffer->setLanguageMode(
        File::getModeForExtension(m_file->getExtension())
    );
    delete oldFile;

    return save();
}

```

Како би се водила евиденција о изменама у тренутној датотеци користи се индикатор који се назива *прљави бит* (енг. *dirty bit*). Он се активира уколико постоје неке измене које нису сачуване. Када корисник отвара датотеку,



прљави бит је неактиван. Сваком променом текста активира се прљави бит, док се сваким чувањем деактивира. Он је обично представљен карактером звездице поред имена тренутно отворене датотеке у статусној линији.

### Отварање нове датотеке

Корисник може отворити нову датотеку притиском на дугме *New file* у менију или комбинацијом тастера *Ctrl* и *N*. Када корисник покрене ову акцију, тренутна датотека се затвара и уколико је активан прљави бит, едитор отвара дијалог у коме корисника пита да ли жели да сачува измене, овај дијалог илустрован је на слици 3.4. Затим се садржај текстуалног поља као и табеле делова ресетује и курсор се поставља на почетак.

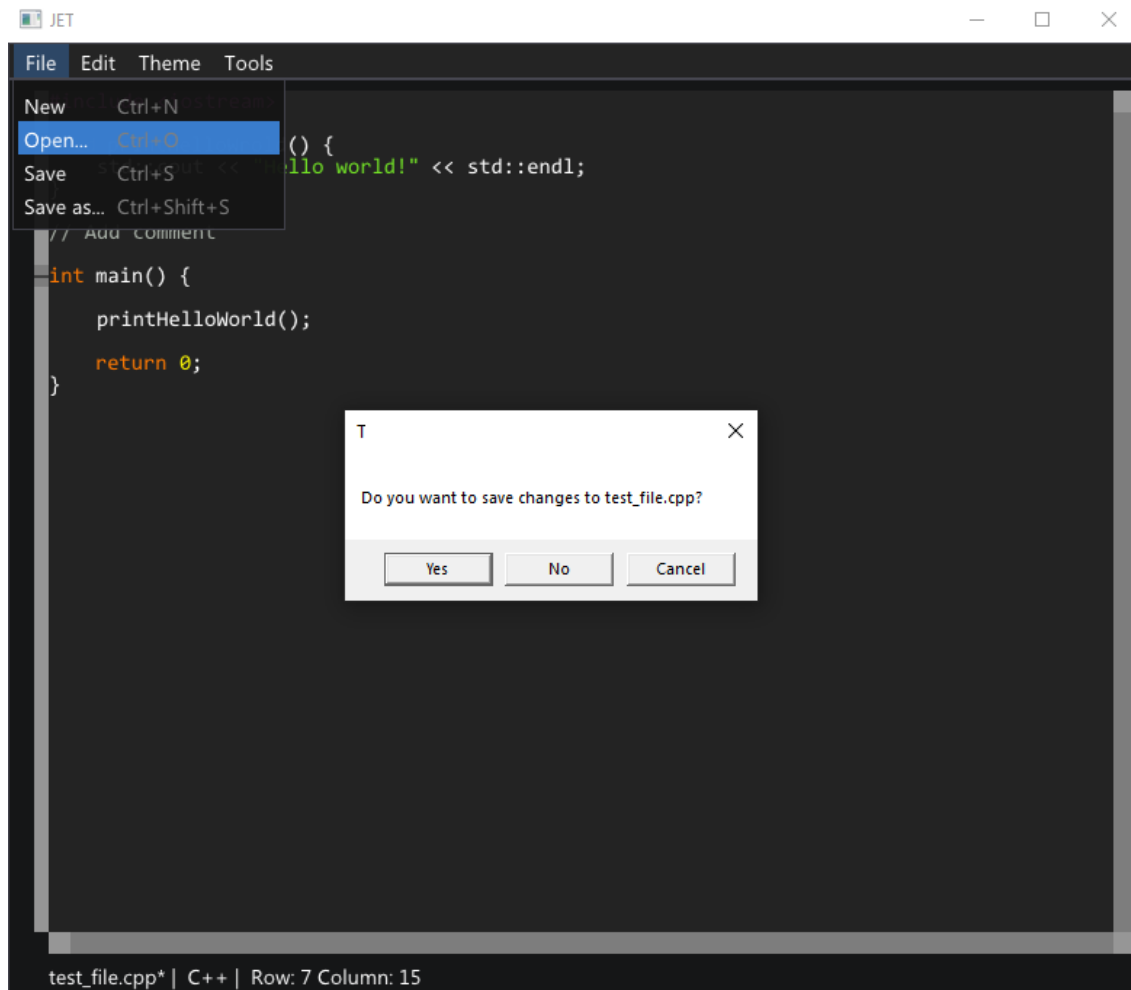
Ова функционалност је имплементирана у оквиру едитора ЈЕТ путем методе `newFile()` класе `TextEditor`. Она проверава да ли је датотека сачувана и позива методу `handleFileNotSaved()` уколико није. Затим позива истоимену методу класе `TextBox` која ресетује свој садржај. У наредном примеру кода приказане су обе описане методе.

```
void TextEditor::newFile() {
    auto id = handleFileNotSaved();
    if (id == IDCANCEL)
        return;

    m_activeTextBox->newFile();
    if (m_splitScreen)
        m_inactiveTextBox->setFile(m_activeTextBox->getFile());
}

void TextBox::newFile() {
    m_pieceTableInstance->newFile();

    m_lineBuffer->getLines();
    m_cursor->clearUndoAndRedoStacks();
    m_cursor->setCoords({1, 1});
    m_scroll->updateScroll(m_width, m_height);
}
```



Слика 3.4: Приказ дијалога за избор датотеке

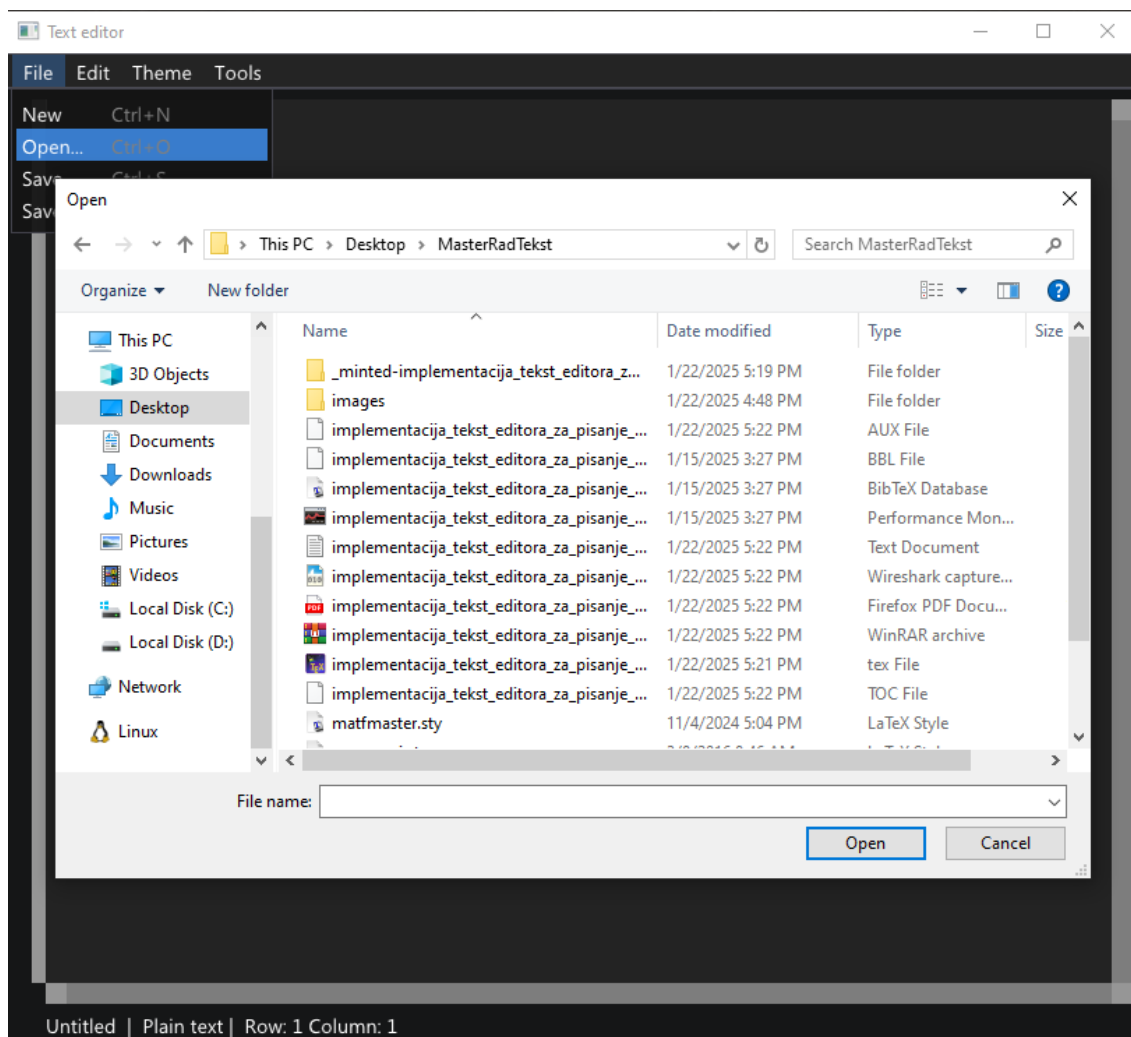
```

    m_scroll->updateMaxScroll(m_width, m_height);
}

```

## Отварање постојеће датотеке

Уколико корисник жели да отвори постојећу датотеку он може то учинити притиском на дугме *Open* у менију или комбинацијом тастера *Ctrl* и *O*. Када покрене ову команду, кориснику се приказује дијалог у коме може да изабере путању до датотеке коју жели да отвори. На слици 3.5 илустрован је дијалог за избор датотеке у едитору текста. Ову функционалност имплементира метода `open()` класе `TextEditor` и илустрована је наредним фрагментом кода.



Слика 3.5: Приказ дијалога за избор датотеке

```
void TextEditor::open() {
    auto id = handleFileNotSaved();
    if (id == IDCANCEL)
        return;

    auto path = openFileDialog();

    if (!path.empty()) {
        m_activeTextBox->open(path);
        m_inactiveTextBox->open(path);
    }
}
```

```
}
```

Прво се проверава прљави бит, и уколико је активан отвара дијалог за чување помоћу функције `handleFileNotSaved()`. Затим се отвара дијалог за избор датотеке помоћу функције `openFileDialog()` и враћа се изабрана путања. На крају се позива истоимена метода класе `TextBox` којом се отвара датотека и њен садржај учитава у едитор.

### 3.6 Исецање, копирање и налепљивање

Исецање, копирање и налепљивање су операције које раде са привременом меморијом. То је део меморије у оперативним системима који је резервисан за привремено чување као и пренос података унутар једног или између више програма. Ти подаци могу бити текст, слике, датотеке, итд. У овом одељку ћемо се бавити преносом текста. Привремена меморија функционише тако што по потреби можемо писати у њу или читати из ње. Обично се у једном тренутку може налазити само један податак у привременој меморији.

Када су ове три операције уведене у едиторе текста током седамдесетих година прошлог века, представљале су велико унапређење у ефикасности писања кода. Корисницима је на тај начин омогућено да ефикасно померају делове текста по потреби као и да копирају неке сличне фрагменте кода који се често користе уместо да их пишу изнова, штедећи доста времена програмерима. У данашње време се едитори текста не могу замислити без ових функционалности, што говори о њиховој корисности и широкој употреби.

#### Исецање

Исецање је операција која копира тренутно означени текст и уписује га у привремену меморију и затим га брише из текста. У наредном примеру кода дата је њена имплементација.

```
void TextBox::cut() {  
    if (m_selection->isActive()) {  
        copySelectionToClipboard();  
        deleteSelection();  
    }  
}
```

```
    }  
}
```

Прво се проверава да ли је неки текст означен: ако јесте копира се њен садржај у привремену меморију, а затим се брише означени текст. У следећем коду дата је имплементација методе `copySelectionToClipboard()`.

```
void TextBox::copySelectionToClipboard() {  
    auto selectionText = m_selection->getSelectionText();  
    ImGui::SetClipboardText(selectionText.c_str());  
}
```

### Копирање

Копирање уписује означени текст у привремену меморију. За разлику од исецања текст се не брише. Имплементација ове операције је приказана у наредном фрагменту кода.

```
void TextBox::copy() {  
    if (m_selection->isActive()) {  
        copySelectionToClipboard();  
    }  
}
```

### Налепљивање

Налепљивање уписује текст из привремене меморије на тренутну позицију курсора. У наредном коду приказана је имплементација ове операције.

```
void TextBox::paste() {  
    auto text = std::string(ImGui::GetClipboardText());  
    if (!text.empty())  
        enterText(text);  
}
```

## Глава 4

# Напредне функционалности

### 4.1 Поништавање и понављање

*Поништавање* (енг. *undo*) је функционалност која поништава последњу акцију над текстом и враћа га у пређашње стање. Дата функционалност пружа могућност кориснику да се врати у неко прошло стање уколико је направљена грешка или обрисан неки битан део текста. Са друге стране, *понављање* (енг. *redo*) поставља текст у стање пре поништавања (уколико је поништена нека акција). Даље објашњење ове функционалности биће илустровано кроз кôд текст едитора JET.

Да би се дефинисало поништавање акција и њихово понављање, прво се морају дефинисати саме акције. Свака измена над текстом у табели делова чини једну акцију. Свака акција се састоји од ознаке да ли се ради о уметању или брисању, индекса на коме почиње измена и низа дескриптора промене. Овај низ садржи редом, с лева на десно, дескрипторе делова који заједно чине додати или обрисани текст. У следећем програмском коду илустроване су чланске променљиве класе `ActionDescriptor`.

```
class ActionDescriptor {
    //...
private:
    ActionType m_actionType;
    std::vector<PieceDescriptor*> m_descriptors;
    size_t m_index;
};
```

У случају уметања текста у табелу делова, прави се акција којој се додељује одговарајућа ознака, индекс на коме је извршено уметање и један дескриптор који садржи додати текст на крају бафера за додавање.

У случају операције брисања индекс се поставља на почетак обрисаног текста, док је низ дескриптора компликованији него код уметања. Пошто се код брисања може обрисати више дескриптора, онда и низ може садржати више елемената. Ако се брише цео дескриптор, онда се исти тај дескриптор копира и додаје у низ. Уколико се дескриптор скраћује са почетка или краја, онда се прави нови дескриптор који показује на обрисани део и додаје се у низ. Исто важи и за случај када се уклања текст из средине једног дескриптора, тј. прави се дескриптор који показује на средину која је обрисана.

Када је познато шта представља једна акција, једноставније је дефинисати како се врши њено поништавање. Наиме, поништавање се врши помоћу *стјека за њоништавање* (енг. *undo stack*) који у себи чува акције. Сваки пут када се изврши нека акција, она се додаје на врх стека. Поништавање се врши тако што се дохвати акција са врха стека, провери се која јој операција одговара, а затим се изврши супротна акција. После тога, дескриптору акције се додељује супротна ознака и додаје се у *стек за њонављање* (енг. *redo stack*).

Поставља се питање како се за произвољну акцију дефинише њој супротна? Супротна акција ће бити дефинисана одвојено за случај уметања и случај брисања текста. Пре него што их дефинишемо, прво ће бити уведене неке ознаке преко којих ће се лакше дефинисати ове операције. Нека је за неку акцију  $a$  њен почетни индекс означен са  $p$ , а њена листа дескриптора делова са  $l$ . Нека је димензија листе  $l$  означена са  $n$ , почетак  $j$ -тог дескриптора листе  $l$  (при чему је  $j \in \{0, 1, \dots, n - 1\}$ ) означавамо са  $p_j$ , а његову дужину са  $d_j$ .

Код уметања текста, супротна акција се извршава тако што се иде редом кроз  $l$  и сабирају се дужине свих њених дескриптора. Нека је сума дужина свих дескриптора  $l$  означена са  $S_d$ , тј.  $S_d = \sum_{j=0}^{n-1} d_j$ . Затим се брише текст из табеле делова који припада целобројном интервалу  $[p, p + S_d)$ .

У случају операције брисања, супротна акција се извршава тако што се пролази редом кроз дескрипторе делова акције и они се умећу у табелу делова

на одговарајућим индексима. Уколико листа  $l$  има димензију  $n$ , то значи да ће бити  $n$  корака уметања. Нека је вредност индекса табеле делова на који вршимо уметање у  $j$ -том кораку означена са  $i_j$ . У том случају ће индекс  $i_j$  имати вредност  $p$  када је  $j = 0$ . Док се  $i_j$  може дефинисати као  $i_j = i_{j-1} + d_{j-1}$ , за случај када важи  $j > 0$ . Операција супротна брисању се изводи тако што се пролази редом кроз листу  $l$  и за  $j$ -ти дескриптор се у табелу делова додаје на индекс  $i_j$  текст из одговарајућег бафера који почиње на индексу  $p_j$  и има дужину  $d_j$ . У наредном програмском коду илустровано је извршавање акције која је супротна акцији са врха стека и њено постављање на други стек.

```
void PieceTable::reverseOperation
(std::stack<ActionDescriptor*>& stack,
 std::stack<ActionDescriptor*>& reverseStack) {

    if (stack.empty())
        return;

    auto action = stack.top();
    stack.pop();

    auto actionType = action->getActionType();
    auto descriptors = action->getDescriptors();
    auto index = action->getIndex();
    size_t totalLength = 0;

    if (actionType == ActionType::Insert) {
        totalLength = std::accumulate(
            descriptors.begin(),
            descriptors.end(),
            (size_t)0,
            [](size_t acc, PieceDescriptor* descriptor)
                { return acc + descriptor->getLength(); }
        );

        deleteText(index, index+totalLength, true);
    } else {
        for (size_t i = 0; i < descriptors.size(); ++i) {
```



```

        auto source = descriptors[i]->getSource();
        auto start = descriptors[i]->getStart();
        auto length = descriptors[i]->getLength();

        insert(source, start, length, index+totalLength, true);
        totalLength += length;
    }
}

auto oppositeActionType =
    ActionDescriptor::getOppositeActionType(actionType);

action->setActionType(oppositeActionType);

reverseStack.push(action);
}

```

Понављање се врши потпуно идентично, само се акција дохвата са стека за понављање, а супротна акција се ставља на стек за поништавање. Може се приметити да су ове операције потпуно симетричне, с тим да се након извршавања неке акције она додаје само на стек за поништавање, док се елементи могу додати на стек за понављање само путем поништавања. У следећем коду је илустрована симетричност ових операција.

```

void PieceTable::undo() {
    reverseOperation(m_undoStack, m_redoStack);
}

void PieceTable::redo() {
    reverseOperation(m_redoStack, m_undoStack);
}

```

Приликом употребе ове функционалност у едиторима текста може се приметити да операције поништавања и понављања обично враћају веће делове текста, а не појединачне карактере. Таква имплементација је интуитивна из угла корисника јер је најчешће потребно поништити веће делове текста уместо један карактер.

Овакав ефекат се постиже тако што се уводе два бафера, један за уметање и један за брисање. Бафер за уметање садржи све карактере који су тренутно унесени у непрекидном низу. Када се низ прекине, било уносом који се не наставља на тренутни низ, било брисањем неког дела текста, бафер се празни и цео његов садржај се уноси у табелу делова, а акција се ставља на стек за поништавање. Бафер за брисање ради по истом принципу као и онај за уметање, само се код њега чува почетни и крајњи индекс обрисаног дела текста.

Све време док бафери нису празни, табела делова даје привид да су све досадашње измене унете у њу, док се у позадини оне уносе тек по пражњењу бафера. У наредна два примера кода илустроване су редом чланске променљиве класе `InsertBuffer` и уношење појединачног карактера у текст.

```
class InsertBuffer {
    //...
private:
    bool m_flushed;
    size_t m_startIndex;
    size_t m_endIndex;
    std::string m_content;
};

bool PieceTable::insertChar(char c, size_t index) {
    if (index != m_insertBuffer->getEndIndex()) {
        flushInsertBuffer();
    }

    bool result = false;

    if (m_insertBuffer->isFlushed()) {
        m_insertBuffer->setStartIndex(index);
        m_insertBuffer->setEndIndex(index);
        m_insertBuffer->setFlushed(false);
        result = true;
    }

    m_insertBuffer->appendToContent(c);
}
```

```
    return result;  
}
```

За крај треба напоменути да је оваква имплементација поништавања и понављања омогућена коришћењем табеле делова. Пошто се текст не брише из бафера могуће је једноставније враћање обрисаних делова текста. Имплементација ових функционалности помоћу других структура података била би значајно сложенија.

## 4.2 Назначавање синтаксе

Назначавање синтаксе (енг. *syntax highlighting*) је напредна функционалност која омогућава кориснику бољу прегледност при писању програмског кода тако што се различити токени као што су ниске, бројеви, кључне речи, коментари, имена функција итд. назначавају различитим бојама. Овим се постиже боља читљивост кода јер корисник лакше разазнаје различите целине у коду.

У оквиру едитора ЈЕТ токени који се назначавају су кључне речи, ниске, бројеви, претпроцесорске директиве и коментари (једнолинијски и вишелинијски). Подржани програмски језици за назначавање су *C++*, *C*, *C#* и *Java*. За сваки језик се чувају наредне информације: да ли има претпроцесорске директиве, како се отварају једнолинијски коментари, како се отварају и затварају вишелинијски коментари и скуп кључних речи. У наредном фрагменту кода илустроване су информације о једном програмском језику у виду чланских променљивих унутар класе `Language`.

```
class Language {  
    //...  
private:  
    std::set<std::string> m_keywords;  
    std::string m_singleLineCommentStart;  
    std::string m_multiLineCommentStart;  
    std::string m_multiLineCommentEnd;  
    std::string m_name;
```

```
bool m_preprocessor;
};
```

Чува се, такође, информација о томе за који програмски језик је укључено назначаваше. Програмски језик се одређује на основу екстензије отворене датотеке, уколико едитор текста за дати програмски језик нема подржано назначаваше синтаксе поставља се режим за обичан текст који не примењује никакво назначаваше за отворену датотеку.

Ова се постиже тако што се за сваки карактер у тексту чува информација о његовој тренутној боји. Она се чува у виду дводимензионе листе где  $j$ -ти елемент  $i$ -те листе одговара  $j$ -том карактеру  $i$ -те линије. Ова тзв. „мапа боја” се ажурира тако што се за различите категорије токена врше провере проласком кроз текст и врши се уписивање одговарајућих вредности у мапу.

Идентификовање претпроцесорских директива се врши провером да ли линија почиње карактером `#` и уколико је то случај цела линија се назначава одговарајућом бојом.

Једнолинијски коментари се препознају тако што се у свакој линији тражи ниска // и уколико је пронађена, она и сав текст десно од ње у текућој линији назначаваче се одговарајућом бојом.

За идентификовање ниски користи се претрага помоћу регуларних израза. У наредном програмском коду приказан је регуларни израз који се користи за претрагу.

```
const std::regex TextHighlighter::m_stringRegex = std::regex(
    R"([\"'])(\\|.|.)*\1)");
```

Бројеви и кључне речи препознају се коришћењем парсера који разбија линију на токене, а затим за сваки токен проверава да ли је симбол и уколико јесте проверава се да ли је садржан у скупу кључних речи. Ако је и овај услов испуњен, токен се означава као кључна реч. У супротном се проверава да ли је токен број и уколико јесте назначаваче одговарајућом бојом. У

наредном примеру кода илустрована је функција која у једној линији тражи све горенаведене токене и враћа исправно обојену мапу за ту линију.

```
std::vector<ThemeColor> TextHighlighter::getColorMap
    (std::string& line, LanguageMode mode) {

    std::vector<ThemeColor> colorMap(line.size(), ThemeColor::TextColor);

    if (LanguageManager::getLanguage(mode)->isPreprocessor())
        searchForPreprocessorCommands(line, colorMap);

    searchRegex(line, colorMap, m_stringRegex, ThemeColor::StringColor);
    searchForSingleLineComment(line, colorMap, mode);
    searchForKeywordsAndNumbers(line, colorMap, mode);

    return colorMap;
}
```

На крају се траже вишелинијски коментари. У свакој линији се тражи отварајућа ознака која је једнака ниски `/*` и када се наиђе на њу пролази се даље кроз текст док се не пронађе затварајућа ознака која је једнака ниски `*/`. Тада се обе ознаке и текст између њих боје у одговарајућу боју и потом се понавља процес тражења отварајуће ознаке. Уколико се при проналажењу неке отварајуће ознаке не пронађе затварајућа онда се текст боји од почетка отварања коментара па до краја самог текста.

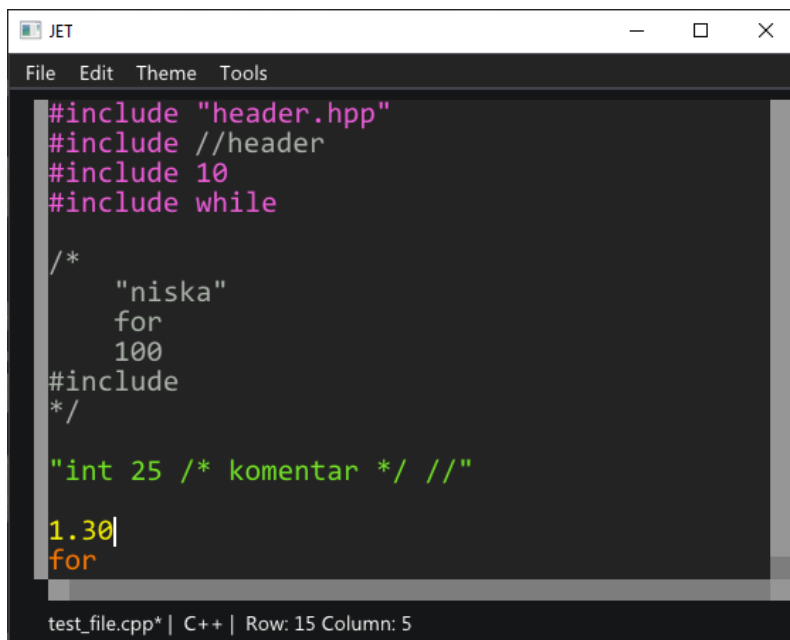
**Пример 4.** *Пример рада овог алгоритма биће илустрован на примеру ниске `/* jedan /* dva */ tri */`. Прво се проналази отварајућа ознака на индексу 0 и она се назначава као коментар и претвара се у наставља десно од ознаке. Затим се тражи затварајућа ознака и проналази се на индексу 16. Све између отварајуће и затварајуће ознаке, укључујући и њу, се боји као коментар и претвара се у наставља после затварајуће ознаке. На крају се тражи отварајућа ознака и пошто није пронађена завршава се са претварањем. На слици 4.1 се може видети назначавање ове ниске у оквиру едитора JET.*

Треба нагласити да различити токени имају различите приоритете тј. уколико неки део текста може бити обојен са више боја треба изабрати само

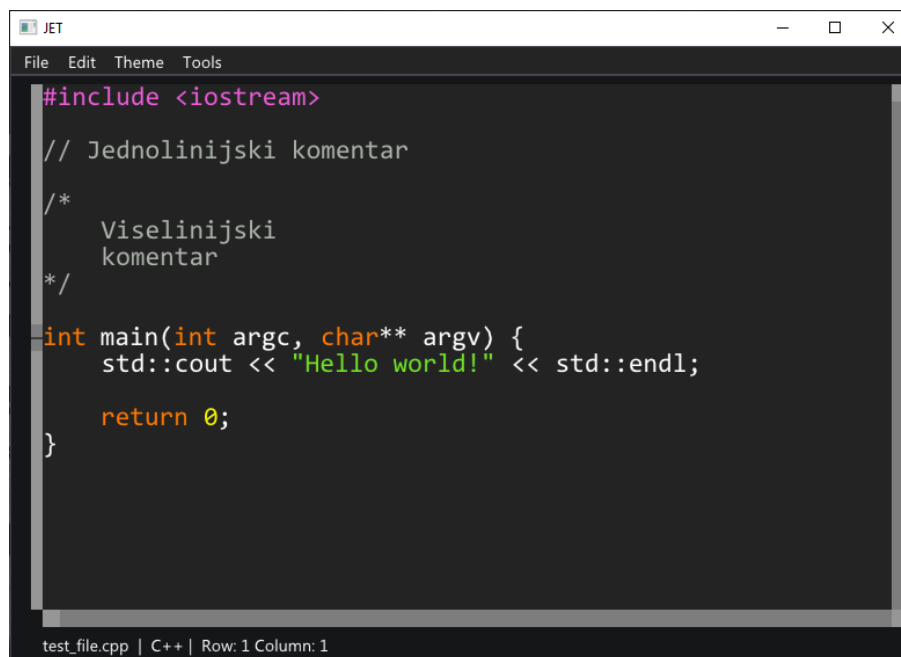


Слика 4.1: Пример назначавања коментара

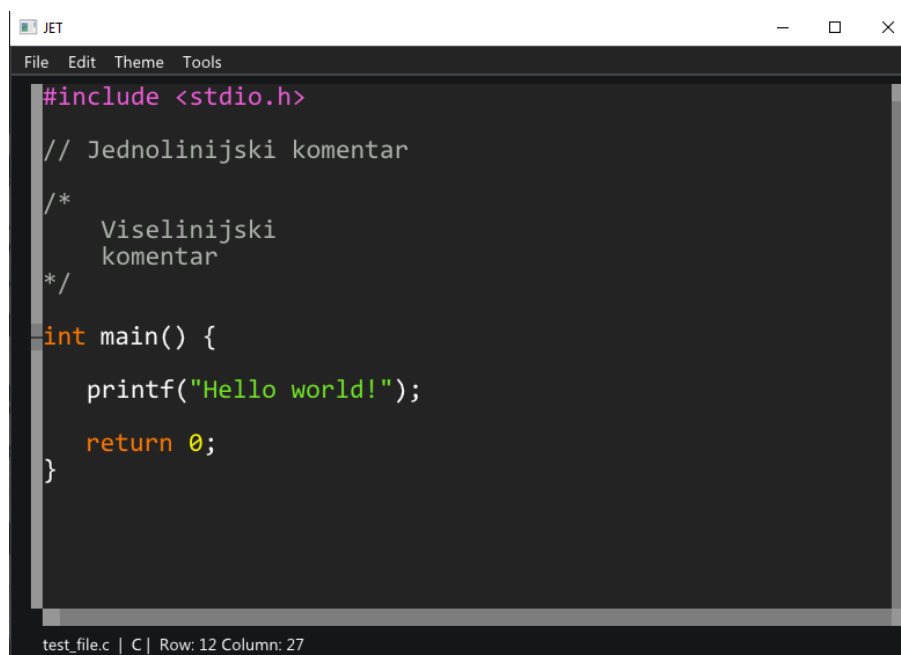
једну. Највиши приоритет имају коментари и ниске, од претпроцесорских директива једино коментари имају већи приоритет, док бројеви и кључне речи имају најнижи приоритет. На слици 4.2 су илустровани приоритети различитих синтаксичких категорија, док је на сликама 4.3, 4.4, 4.5 и 4.6 илустровано редом назначавање програмског кода за језике *C++*, *C*, *C#* и *Java*. Претпроцесорске директиве обојене су у розе, коментари сивом бојом, ниске су обојене у светло зелено, бројеви жутом бојом, док су кључне речи обојене наранџасто.



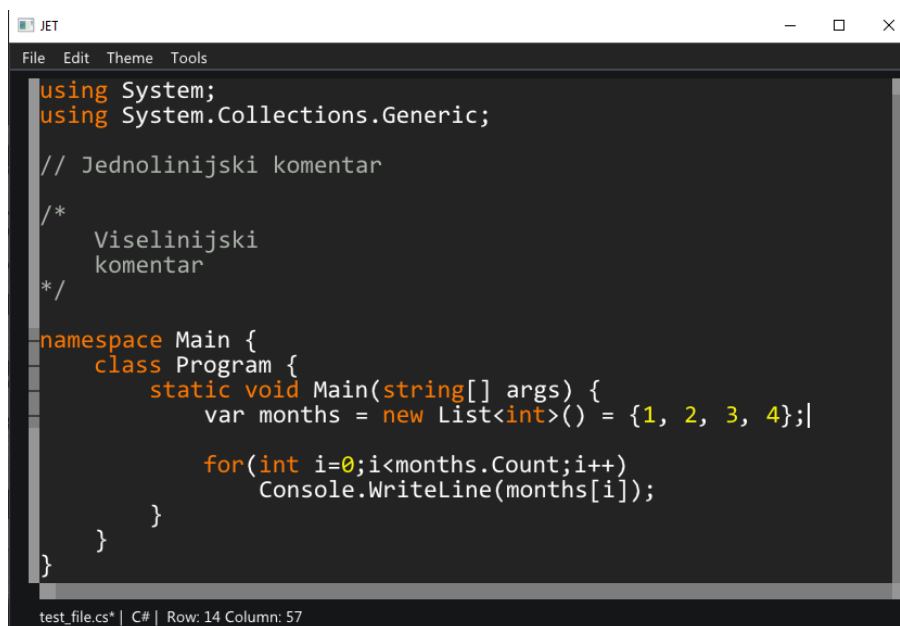
Слика 4.2: Приказ приоритета између различитих синтаксичких категорија у програмском језику *C++*



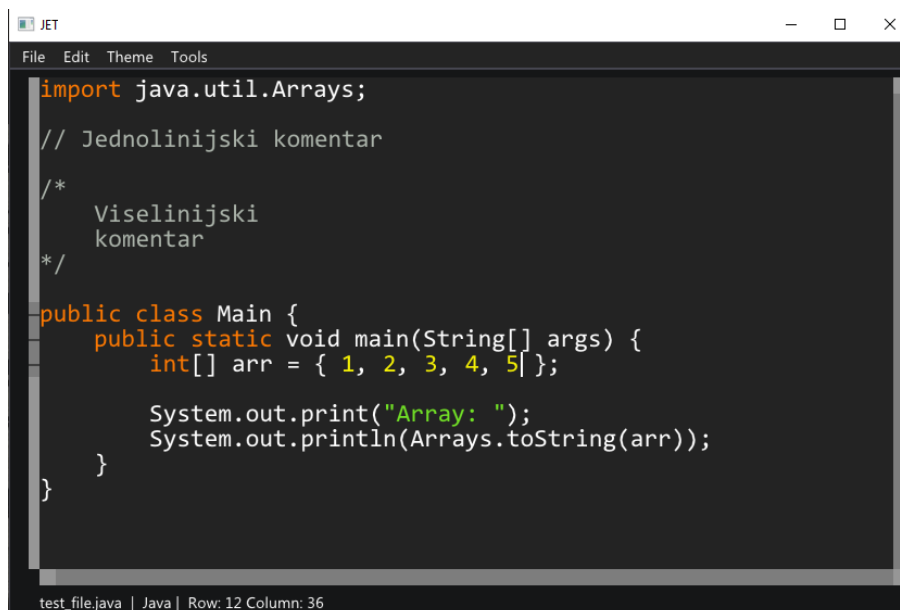
Слика 4.3: Приказ назначаванја кода у програмском језику *C++*



Слика 4.4: Приказ назначаванја кода у програмском језику *C*



Слика 4.5: Приказ назначавања кода у програмском језику *C#*



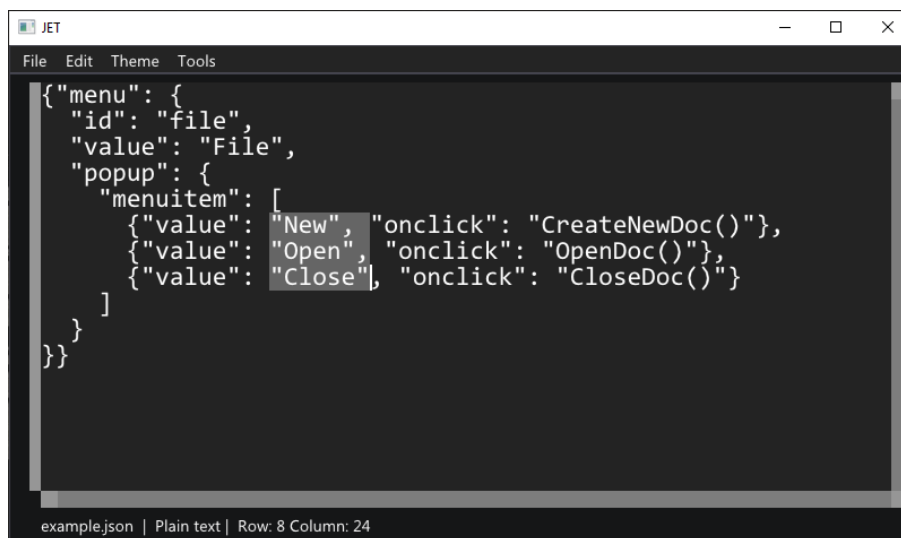
Слика 4.6: Приказ назначавања кода у програмском језику *Java*

### 4.3 Правоугаоно означавање

У општем случају, означени текст представља непрекидни део текста између почетне и крајње координате. Међутим, *правоугаоно означавање* омо-



гућава да се изабере само онај део текста који је унутар правоугаоника који чине почетна и крајња координата.



Слика 4.7: Приказ правоугаоног означавања

То се постиже тако што се додаје индикатор који нам говори да ли је означавање обично или је правоугаоно. При дохватању означеног текста, прво се проверава вредност индикатора. Уколико се ради о обичном означавању, две координате се преводе у индексе и дохвата се текст између њих. Уколико се ради о правоугаоном означавању, онда се за сваку линију почевши од линије почетне координате па до крајње, дохвата текст између колона почетне и крајње координате. У наредном програмском коду је илустрована ова функционалност.

```
std::string Selection::getSelectionText() {
    std::string result;
    TextCoordinates it = m_start.getCoords();

    // Dohvata se pokazivač na početnu liniju
    std::string* line = &m_lineBuffer->lineAt(it.m_row-1);

    // Iteriranje dok se ne stigne do kraja selekcije
    while (it < m_end.getCoords()) {
        // Ako je kolona van granica, ide se na narednu liniju
```

```
        if (it.m_col > line->size())
        || (m_rectangular && it.m_col >= m_end.getCoords().m_col)) {
            result += '\n';
            it.m_row += 1;
            it.m_col = m_rectangular ? m_start.getCoords().m_col : 1;

            line = &m_lineBuffer->lineAt(it.m_row-1);
        } else {
            if (!line->empty())
                result.push_back(line->at(it.m_col-1));

            it.m_col += 1;
        }
    }

    return result;
}
```

Ова функционалност омогућава једноставно дохватање испрекиданих делова текста као што су колоне у текстуалним табелама као и делове кода без дохватања карактера за назубљивање. На слици 4.7 илустровано је правоугаоно означавање где је означен атрибут типа `value`.

## 4.4 Подела екрана

Дељење екрана пружа кориснику могућност да може истовремено да прегледа и/или да мења исту датотеку, поделом екрана на два поља за уређивање. Корисник укључује и искључује овај режим преко менија у зависности од потребе. Ова функционалност је корисна у случајевима када текстуална датотека садржи велики број линија и корисник може лакше да прегледа или мења два удаљена места у тексту без сталног пребацивања између њих.

Ова функционалност је едитору ЈЕТ остварена тако што се цртају два објекта класе `TextBox` који деле показивач на исту табелу делова. Осим што садрже исти показивач на табелу делова, која је имплементирана у класи `PieceTable`, све чланске променљиве две инстанце класе `TextBox` су разли-

чите и показују на различите објекте. На слици 4.8 је илустрована функционалност дељења екрана за једну датотеку.



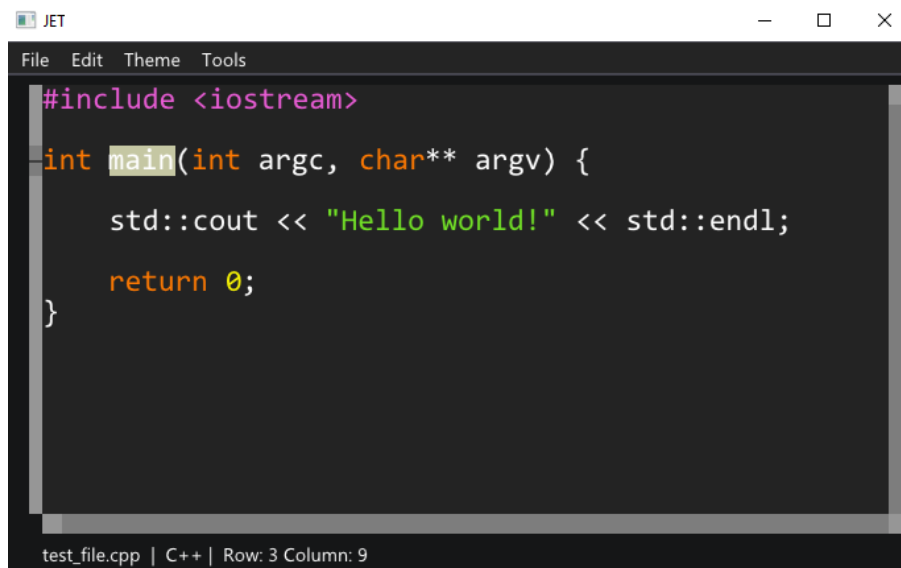
Слика 4.8: Приказ поделе екрана

### 4.5 Сужавање простора за измену

Сужавање простора за измену омогућава да се место на коме корисник жели да изврши измене сузи на неки произвољни опсег текста. Корисник означава неки део текста, активира ову функционалност из менија и од тада је могуће вршити измене само унутар означеног дела. Корисник у сваком тренутку може искључити сужавање, враћајући се у уобичајени режим. Ова функционалност може бити корисна у случајевима када је потребно мењати мање делова кода без нарушавања околне структуре. Пример овакве измене може бити промена имена функције или променљиве.

Ова функционалност је у едитору ЈЕТ имплементирана тако што се при активирању овог режима памте почетак и крај означеног текста, док у случају ако није означен текст није могуће активирати овај режим. Затим се кретање курсора и даље означавање текста ограничава на сужени део за измене. То

се постиже тако што се за сваку нову позицију курсора врши провера да ли се налази у дозвољеним границама и уколико то није случај поставља се на најближу граничну вредност. Део за измене се, такође, сужава или проширује у зависности од тога да ли се брише или додаје текст у суженом простору. На слици 4.9 илустровано је сужавање дела за измену на име главне функције.



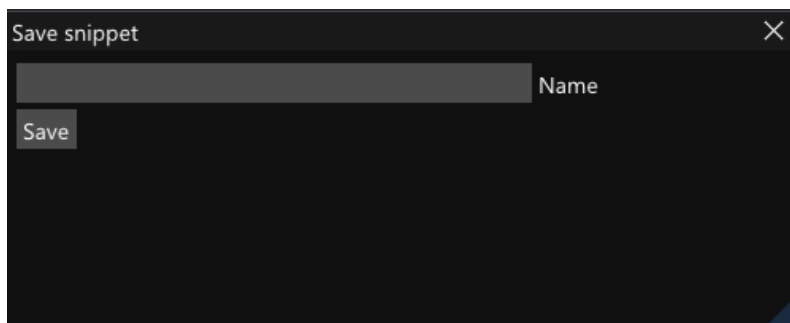
Слика 4.9: Приказ сужавања дела за измену

### 4.6 Чување исечака кода

*Исечак кода* (енг. *code snippet*) представља фрагмент кода који се може користити у различитим програмима или деловима једног програма. Исечци се обично чувају у некој датотеци јер представљају корисне делове кода који се могу користити касније и нема потребе за њиховим поновним писањем.

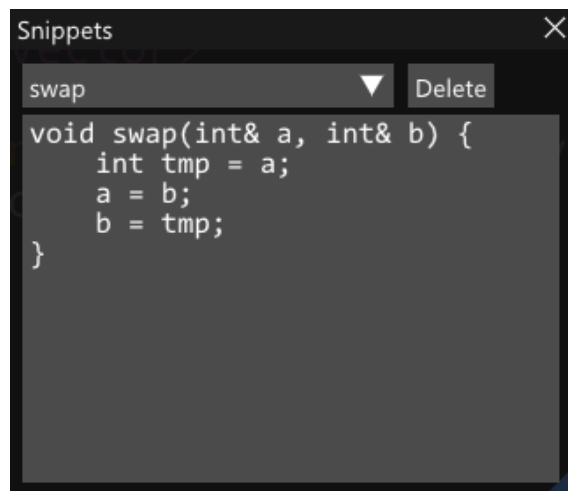
Чување исечака кода омогућава кориснику да сачува исечке кода које често користи при писању програмског кода. Сваки исечак састоји се од имена и његовог садржаја. Списку свих запамћених исечака се може приступити путем дијалога за исечке. Након одабира жељеног исечка кода корисник може прекопирати његов текст или га обрисати у зависности од потребе. Корисник може додати нови исечак означавањем жељеног текста и одабиром акције додавања исечка из менија, при чему се отвара дијалог где се од

корисника тражи да унесе име новог исечка и уколико не постоји исечак са истим именом, он се додаје у колекцију. Дијалог за одабир имена исечка је илустрован на слици 4.10.



Слика 4.10: Приказ дијалога за одабир имена исечака

Описана функционалност је постигнута тако што се исечци чувају у датотекама које се по потреби дохватају и њихов садржај се исписује у дијалогу. Овај дијалог је илустрован на слици 4.11.



Слика 4.11: Приказ дијалога за одабир исечака

## 4.7 Сакривање блокова

*Сакривање блокова* (енг. *code folding*) је функционалност која кориснику омогућава да сажме приказ блока кода у једну линију. Корисник може по

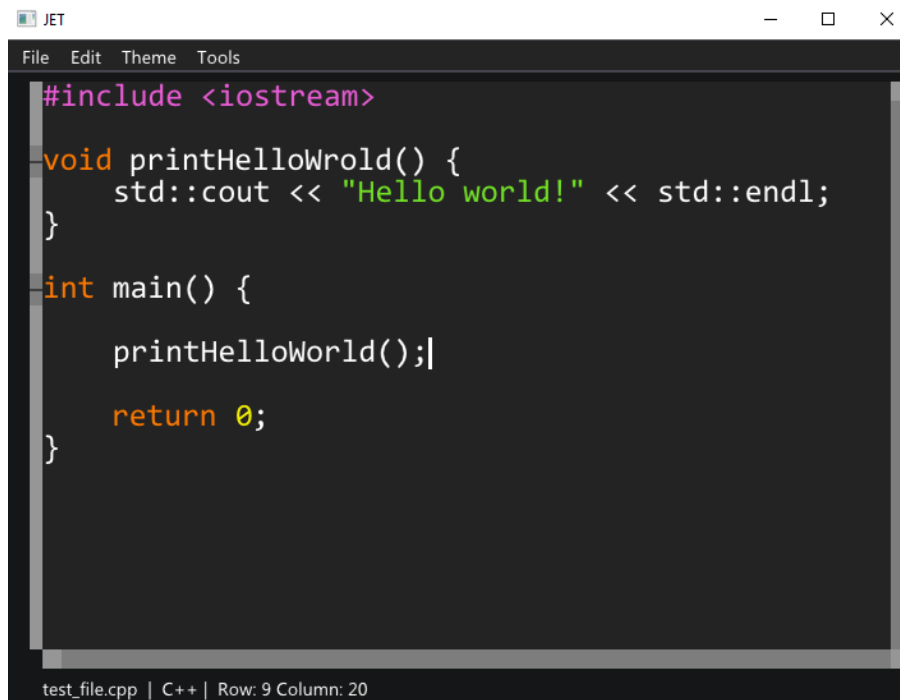
потреби да сакрива блокове кода који му у том тренутку нису релевантни и тако добије бољу прегледност унутар датотеке. Пре него што наставимо са даљим описом сакривања блокова, потребно је дефинисати шта представља блок у неком програмском коду.

Блок кода је скуп наредби које су логички груписане и чине јединствену целину. Блокови се најчешће користе за дефинисање функција, наредби контроле тока и петљи. У већини програмских језика (*C++*, *C*, *C#*, *Java*) блокови се означавају витичастим заградама, док се код неких, као што је *Python*, они означавају назубљивањем кода. У оквиру едитора *JET* блок се састоји од почетне и крајње координате као и индикатора да ли је блок сакривен или не. У наредном програмском коду илустроване су чланске променљиве класе `CodeBlock`.

```
class CodeBlock {  
    //...  
private:  
    TextCoordinates m_start;  
    TextCoordinates m_end;  
    bool m_folded;  
};
```

У едитору текста *JET*, сакривање блока је имплементирано тако што се пролази кроз текст и идентификују се сви блокови. То се постиже тако што се пролази кроз текст и сваки пут када се наиђе на карактер `{`, његове координате се стављају на врх стека за отворене заграде. Када се наиђе на карактер `}` онда се гледа да ли је стек празан и уколико није, узима се елемент са врха стека и заједно са координатама тренутног карактера креира се нови блок. Потом се у првој линији сваког блока са леве стране додаје дугме за сакривање. Знак минус означава да се притиском на дугме одговарајући блок сакрива, а знак плус да се притиском на њега блок открива. Пример сакривања блока је илустрован на сликама 4.12 и 4.13, где је сакривен блок функције `printHelloWorld()`.

Евиденција о скривеним линијама кода се чува у низу индикатора који за сваку линију садржи вредност да ли је скривена или не. Уколико је линија



```
#include <iostream>

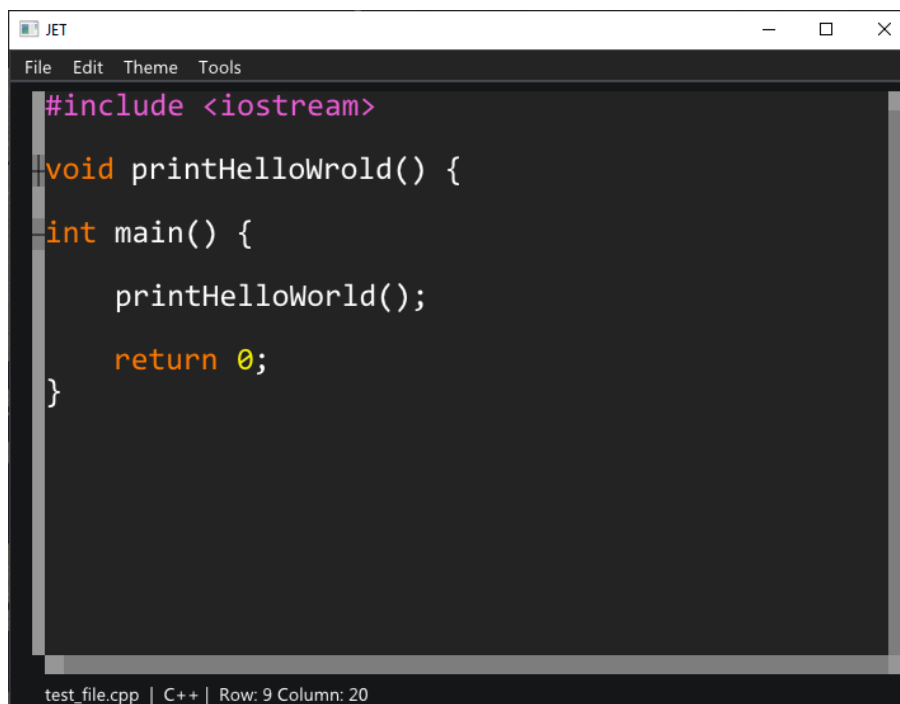
void printHelloWorld() {
    std::cout << "Hello world!" << std::endl;
}

int main() {
    printHelloWorld();

    return 0;
}
```

test\_file.cpp | C++ | Row: 9 Column: 20

Слика 4.12: Пример сакривања блока програмског кода



```
#include <iostream>

void printHelloWorld() {
int main() {
    printHelloWorld();

    return 0;
}
```

test\_file.cpp | C++ | Row: 9 Column: 20

Слика 4.13: Пример сакривања блока програмског кода

скривена, не исцртава се у текстуалном пољу, иако и даље постоји унутар текста.

При свакој промени текста потребно је поново идентификовати блокове. Може се десити да промена текста помери координате постојећих блокова и потребно је да у поновној идентификацији буду препознати као постојећи блокови, а не као нови. То се постиже тако што се уз сваку промену текста функцији за ажурирање блокова прослеђује линија на којој је извршена измена, као број додатих или одузетих линија. Функција за ажурирање блокова, пре него што обради нове блокове, помериће координате старих блокова који су захваћени променом текста за одговарајући размак. На крају се за сваки нови блок, бинарном претрагом, у листи старих блокова тражи одговарајући блок, и уколико постоји, преписује се његово стање у нови. На крају се брише листа старих блокова и прослеђује се нова листа. У следећем програмском коду је илустрован рад ове функције.

```
void LineBuffer::updateBlocks(int lineSizeDiff, int lineIndex) {
    auto newBlocks = new std::vector<CodeBlock*>();
    std::stack<TextCoordinates> openBrackets;
    auto size = m_lines->size();

    if (lineSizeDiff != 0)
        updateFoldedBlocks(lineSizeDiff, lineIndex);

    auto oldHidden = m_hidden;
    m_hidden = new std::vector<bool>(size, false);
    delete oldHidden;

    for (size_t i=0; i<size; ++i) {
        auto line = m_lines->at(i);

        for (size_t j=0; j<line.size(); ++j) {
            if (line[j] == '{') {
                openBrackets.push(TextCoordinates(i, j+1));
            } else if (line[j] == '}' && !openBrackets.empty()) {
                auto openBracket = openBrackets.top();
                openBrackets.pop();
            }
        }
    }
}
```



```

        auto closedBracket = TextCoordinates(i, j+1);
        auto block = new CodeBlock(openBracket, closedBracket);

        auto index = findBlock(block);

        if (index != -1 && m_blocks->at(index)->isFolded() == true) {
            block->setFolded(true);
            updateHiddenForBlock(block);
        }

        newBlocks->push_back(block);
    }
}

auto oldBlocks = m_blocks;
m_blocks = newBlocks;

for (auto block : *oldBlocks)
    delete block;
delete oldBlocks;

std::sort(m_blocks->begin(), m_blocks->end(),
    [](CodeBlock* first, CodeBlock* second) { return *first < *second; }
);
}

void LineBuffer::updateFoldedBlocks(int lineSizeDiff, int lineIndex) {
    if (lineIndex == -1)
        return;

    if (lineSizeDiff < 0)
        lineIndex -= lineSizeDiff;

    auto index = findGreaterOrEqualBlock(lineIndex);

```

```

while (index != -1 && index < m_blocks->size()) {
    auto block = m_blocks->at(index);

    if (block->isFolded()) {
        block->setStart({block->getStart().m_row + lineSizeDiff, block->getStart().m_col});
        block->setEnd({block->getEnd().m_row + lineSizeDiff, block->getEnd().m_col});
    }

    ++index;
}

}

int LineBuffer::findBlock(CodeBlock *block) {
    int low = 0;
    int high = m_blocks->size()-1;
    int mid;

    while (high >= low) {
        mid = low + (high - low) / 2;

        if (*(m_blocks->at(mid)) == *block)
            return mid;

        if (*(m_blocks->at(mid)) > *block)
            high = mid-1;
        else
            low = mid+1;
    }
    return -1;
}

int LineBuffer::findGreaterOrEqualBlock(size_t lineIndex) {
    int result = -1;

    int low = 0;
    int high = m_blocks->size()-1;
    int mid;

```

```
while (low <= high) {
    mid = low + (high - low) / 2;

    if (m_blocks->at(mid)->getStart().m_row >= lineIndex) {
        result = mid;
        high = mid-1;
    } else {
        low = mid+1;
    }
}

return result;
}
```

Још једна битна особина код сакривања блокова је угнежђеност. Пошто унутар једног блока можемо дефинисати прозивољан број других блокова, треба приметити да када је један блок сакривен, сви блокови унутар њега се такође не приказују, без обзира на то што њихов индикатор сакривености остаје непромењен. Са друге стране, уколико је блок откривен, сви блокови унутар њега се приказују на основу својих индикатора и правило се рекурзивно примењује на све откривене блокове у њему.

## Глава 5

# Закључак

У овом раду извршен је преглед имплементације једног едитора текста. Почевши од структура података које се користе за манипулацију над текстом, стечен је увид у сложеност њихових имплементација, као и у њихову временску ефикасност у односу на једноставне приступе.

У наставку рада обрађене су основне функционалности едитора текста. Прво је описана обрада уноса, како сигнала које корисник задаје помоћу тастатуре, тако и оних који задаје помоћу миша. Уведен је појам курсора, који омогућава кориснику да се креће по тексту и мења различите делове текста. Описано је како се додају или бришу појединачни карактери у тексту, као и манипулација непрекидних делова текста кроз селекцију и операције са меморијском таблом. Поред тога, обрађено је и управљање текстуалним датотекама које корисник може да креира, или да чита и мења њихов садржај.

На крају су описане напредне функционалности едитора текста. Описан је сложени механизам поништавања и понављања који се данас налази готово у сваком едитору текста. Обрађено је назначавање синтаксе за различите програмске језике које омогућава боље разумевање целина у програмском коду. Потом су описане неке нестандартне функционалности као што су правоугаоно означавање текста, корисно за копирање више линија текста које су назубљене, подела екрана, сужавање простора за измену као и чување исечака кода. Док је у последњем одељку описан поступак сакривања блокова програмског кода ради боље прегледности.

Кроз овај рад се може увидети сложеност софтвера какви су едитори текста проласком кроз бројне функционалности потребне за њихову израду. Улазећи у детаље имплементације сваке од ових функционалности, упознајемо се са њиховим функционисањем на дубоком нивоу, као и са функционисањем едитора текста у целини.

# Библиографија

- [1] Visual Studio Code. <https://code.visualstudio.com/>.
- [2] Open Source Community. Emacs. <https://www.gnu.org/software/emacs/>.
- [3] Open Source Community. Notepad++. <https://notepad-plus-plus.org/>.
- [4] Open Source Community. Xi. <https://xi-editor.io/>.
- [5] Microsoft. Windows Notepad. [https://en.wikipedia.org/wiki/Windows\\_Notepad](https://en.wikipedia.org/wiki/Windows_Notepad).
- [6] Bram Moolenaar. VIM. <https://www.vim.org/>.

# Биографија аутора

**Бојан Барџић** је рођен 5. априла 1999. године у Призрену. Основне студије је завршио на Математичком факултету у Београду, на смеру информатика 2022. године са просечном оценом 8.64. Тренутно је студент мастер студија на истом факултету.