Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ КОМП'ЮТЕРНИХ НАУК ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Катедра «Комп'ютерна інженерія та програмування»

ЗВІТ

про виконання лабораторної роботи №6

з навчальної дисципліни «Алгоритми та структури даних»

Варіант 5

Виконав студент:

Омельніцький Андрій Миколайович
Група: КН-1023б

Перевірив:

Старший викладач

Бульба Сергій Сергійович

Харків - 2024

Зміст

1 Мета роботи

Мета: Отримати практичні навички та закріпити знання про можливі подання даних типу рядок та про операції над рядками. Теми для попередньої роботи:
Теми для попередньої роботи:

- подання рядків;
- операції над рядками;
- засоби обробки рядків у мовах програмування;
- текстові процесори.

2 Завдання

Написати програму, в якій передбачити виконання вказаної операції над рядками за умови подання рядків у пам’яті двома способами. Порівняти подання рядків вказаними способами за такими показниками:

- обсягом використовуваної пам’яті;
- часом виконання функції;

Спосіб подання рядка:

- Вектор з керованою довжиною рядка (з дескриптором).
- Блочно-зв’язне подання із керованою довжиною.

Функція: скопіювати з рядка s, починаючи з m-го символу, n символів.

3 Хід виконання

Для виконання завдання було обрано мову Rust. Увесь код також додатково був розміщений в GitHub репозитарії: https://github.com/blackgolyb/algos-labs.

3.1 Vector

Для цього завдання напишемо власну реалізацю вектора яка буде використовуватися далі

```rust
use std::alloc::{self, Layout};
use std::marker::PhantomData;
use std::mem;
use std::ops::{Deref, DerefMut};
use std::ptr::{self, NonNull};

use super::super::utils::clamp_range;

#[derive(Debug)]
struct RawVector<T> {
    ptr: NonNull<T>,
    cap: usize,
}

unsafe impl<T: Send> Send for RawVector<T> {}
unsafe impl<T: Sync> Sync for RawVector<T> {}

impl<T> RawVector<T> {
    fn new() -> Self {
        // !0 is usize::MAX. This branch should be stripped at compile time.
        let cap = if mem::size_of::<T>() == 0 { !0 } else { 0 };

        // `NonNull::dangling()` doubles as "unallocated" and "zero-sized allocation"
        RawVector {
            ptr: NonNull::dangling(),
            cap,
        }
    }

    fn grow(&mut self) {
        self.grow_with_capacity(None);
    }

    fn grow_with_capacity(&mut self, capacity: Option<usize>) {
        // since we set the capacity to usize::MAX when T has size 0,
        // getting to here necessarily means the Vector is overfull.
        assert!(mem::size_of::<T>() != 0, "capacity overflow");

        let (new_cap, new_layout) = if let Some(capacity) = capacity {
            assert!(
                capacity.unwrap() > self.cap,
                "new capacity must be bigger than current capacity"
            );

            let new_cap = capacity.unwrap();

            let new_layout = Layout::array::<T>(new_cap).unwrap();
            (new_cap, new_layout)
        } else if self.cap == 0 {
            (1, Layout::array::<T>(1).unwrap())
        } else {
            // This can't overflow because we ensure self.cap <= isize::MAX.
            let new_cap = 2 * self.cap;

            // `Layout::array` checks that the number of bytes is <= usize::MAX,
            // but this is redundant since old_layout.size() <= isize::MAX,
            // so the `unwrap` should never fail.
            let new_layout = Layout::array::<T>(new_cap).unwrap();
            (new_cap, new_layout)
        };

        // Ensure that the new allocation doesn't exceed `isize::MAX` bytes.
        assert!(
            new_layout.size() <= isize::MAX as usize,
            "Allocation too large"
        );

        let new_ptr = if self.cap == 0 {
            unsafe { alloc::alloc(new_layout) }
```

```
70          } else {
71              let old_layout = Layout::array::<T>(self.cap).unwrap();
72              let old_ptr = self.ptr.as_ptr() as *mut u8;
73              unsafe { alloc::realloc(old_ptr, old_layout, new_layout.size()) }
74          };
75
76          // If allocation fails, `new_ptr` will be null, in which case we abort.
77          self.ptr = match NonNull::new(new_ptr as *mut T) {
78              Some(p) => p,
79              None => alloc::handle_alloc_error(new_layout),
80          };
81          self.cap = new_cap;
82      }
83
84      fn slice(&self, start: usize, end: usize) -> Self {
85          let len = end - start;
86          let mut new_vec = Self::new();
87          new_vec.grow_with_capacity(Some(len));
88
89          unsafe {
90              ptr::copy_nonoverlapping(
91                  self.ptr.as_ptr().add(start),
92                  new_vec.ptr.as_ptr(),
93                  len,
94              );
95          }
96
97          new_vec
98      }
99  }
100
101 impl<T> Drop for RawVector<T> {
102     fn drop(&mut self) {
103         let elem_size = mem::size_of::<T>();
104
105         if self.cap != 0 && elem_size != 0 {
106             unsafe {
107                 alloc::dealloc(
108                     self.ptr.as_ptr() as *mut u8,
109                     Layout::array::<T>(self.cap).unwrap(),
110                 );
111             }
112         }
113     }
114 }
115
116 impl<T: Clone> Clone for RawVector<T> {
117     fn clone(&self) -> Self {
118         let mut new_vec = RawVector::new();
119         if self.cap != 0 {
120             new_vec.grow_with_capacity(Some(self.cap));
121
122             unsafe {
123                 ptr::copy_nonoverlapping(self.ptr.as_ptr(), new_vec.ptr.as_ptr(), self.cap);
124             }
125         }
126
127         new_vec
128     }
129 }
130
131 #[derive(Debug, Clone)]
132 pub struct Vector<T> {
133     buf: RawVector<T>,
134     len: usize,
135 }
136
137 impl<T> Vector<T> {
138     fn ptr(&self) -> *mut T {
139         self.buf.ptr.as_ptr()
140     }
141
142     fn cap(&self) -> usize {
143         self.buf.cap
144     }
145
```

```
146     pub fn new() -> Self {
147         Vector {
148             buf: RawVector::new(),
149             len: 0,
150         }
151     }
152
153     pub fn with_capacity(capacity: usize) -> Self {
154         let mut v = Self::new();
155         if capacity > 0 {
156             v.buf.grow_with_capacity(Some(capacity));
157         }
158         v
159     }
160
161     pub fn grow_to(&mut self, capacity: usize) {
162         self.buf.grow_with_capacity(Some(capacity));
163     }
164
165     pub fn push(&mut self, elem: T) {
166         if self.len == self.cap() {
167             self.buf.grow();
168         }
169
170         unsafe {
171             ptr::write(self.ptr().add(self.len), elem);
172         }
173
174         // Can't overflow, we'll OOM first.
175         self.len += 1;
176     }
177
178     pub fn pop(&mut self) -> Option<T> {
179         if self.len == 0 {
180             None
181         } else {
182             self.len -= 1;
183             unsafe { Some(ptr::read(self.ptr().add(self.len))) }
184         }
185     }
186
187     pub fn insert(&mut self, index: usize, elem: T) {
188         assert!(index <= self.len, "index␣out␣of␣bounds");
189         if self.len == self.cap() {
190             self.buf.grow();
191         }
192
193         unsafe {
194             ptr::copy(
195                 self.ptr().add(index),
196                 self.ptr().add(index + 1),
197                 self.len - index,
198             );
199             ptr::write(self.ptr().add(index), elem);
200         }
201
202         self.len += 1;
203     }
204
205     pub fn remove(&mut self, index: usize) -> T {
206         assert!(index < self.len, "index␣out␣of␣bounds");
207
208         self.len -= 1;
209
210         unsafe {
211             let result = ptr::read(self.ptr().add(index));
212             ptr::copy(
213                 self.ptr().add(index + 1),
214                 self.ptr().add(index),
215                 self.len - index,
216             );
217             result
218         }
219     }
220
221     pub fn drain(&mut self) -> Drain<T> {
```

```
222             let iter = unsafe { RawValIter::new(&self) };
223
224             // this is a mem::forget safety thing. If Drain is forgotten, we just
225             // leak the whole Vector's contents. Also we need to do this *eventually*
226             // anyway, so why not do it now?
227             self.len = 0;
228
229             Drain {
230                 iter,
231                 vector: PhantomData,
232             }
233         }
234
235         pub fn slice(&self, start: i64, end: i64) -> Self {
236             let (start, end) = clamp_range(self.len(), start, end);
237
238             if start >= end {
239                 return Self::new();
240             }
241
242             let vec = self.buf.slice(start as usize, end as usize);
243
244             Self {
245                 buf: vec,
246                 len: (end - start) as usize,
247             }
248         }
249
250         pub fn full_size(&self) -> usize {
251             self.buf.cap * mem::size_of::<T>() + mem::size_of::<Self>()
252         }
253     }
254
255     impl<T> Drop for Vector<T> {
256         fn drop(&mut self) {
257             while let Some(_) = self.pop() {}
258             // deallocation is handled by RawVector
259         }
260     }
261
262     impl<T> Deref for Vector<T> {
263         type Target = [T];
264         fn deref(&self) -> &[T] {
265             unsafe { std::slice::from_raw_parts(self.ptr(), self.len) }
266         }
267     }
268
269     impl<T> DerefMut for Vector<T> {
270         fn deref_mut(&mut self) -> &mut [T] {
271             unsafe { std::slice::from_raw_parts_mut(self.ptr(), self.len) }
272         }
273     }
274
275     impl<T> IntoIterator for Vector<T> {
276         type Item = T;
277         type IntoIter = IntoIter<T>;
278         fn into_iter(self) -> IntoIter<T> {
279             let (iter, buf) = unsafe { (RawValIter::new(&self), ptr::read(&self.buf)) };
280
281             mem::forget(self);
282
283             IntoIter { iter, _buf: buf }
284         }
285     }
286
287     struct RawValIter<T> {
288         start: *const T,
289         end: *const T,
290     }
291
292     impl<T> RawValIter<T> {
293         unsafe fn new(slice: &[T]) -> Self {
294             RawValIter {
295                 start: slice.as_ptr(),
296                 end: if mem::size_of::<T>() == 0 {
297                     ((slice.as_ptr() as usize) + slice.len()) as *const _
```

```
298                 } else if slice.len() == 0 {
299                     slice.as_ptr()
300                 } else {
301                     slice.as_ptr().add(slice.len())
302                 },
303             }
304         }
305     }
306
307     impl<T> Iterator for RawValIter<T> {
308         type Item = T;
309         fn next(&mut self) -> Option<T> {
310             if self.start == self.end {
311                 None
312             } else {
313                 unsafe {
314                     if mem::size_of::<T>() == 0 {
315                         self.start = (self.start as usize + 1) as *const _;
316                         Some(ptr::read(NonNull::<T>::dangling().as_ptr()))
317                     } else {
318                         let old_ptr = self.start;
319                         self.start = self.start.offset(1);
320                         Some(ptr::read(old_ptr))
321                     }
322                 }
323             }
324         }
325
326         fn size_hint(&self) -> (usize, Option<usize>) {
327             let elem_size = mem::size_of::<T>();
328             let len =
329                 (self.end as usize - self.start as usize) / if elem_size == 0 { 1 } else { elem_size };
330             (len, Some(len))
331         }
332     }
333
334     impl<T> DoubleEndedIterator for RawValIter<T> {
335         fn next_back(&mut self) -> Option<T> {
336             if self.start == self.end {
337                 None
338             } else {
339                 unsafe {
340                     if mem::size_of::<T>() == 0 {
341                         self.end = (self.end as usize - 1) as *const _;
342                         Some(ptr::read(NonNull::<T>::dangling().as_ptr()))
343                     } else {
344                         self.end = self.end.offset(-1);
345                         Some(ptr::read(self.end))
346                     }
347                 }
348             }
349         }
350     }
351
352     pub struct IntoIter<T> {
353         _buf: RawVector<T>, // we don't actually care about this. Just need it to live.
354         iter: RawValIter<T>,
355     }
356
357     impl<T> Iterator for IntoIter<T> {
358         type Item = T;
359         fn next(&mut self) -> Option<T> {
360             self.iter.next()
361         }
362         fn size_hint(&self) -> (usize, Option<usize>) {
363             self.iter.size_hint()
364         }
365     }
366
367     impl<T> DoubleEndedIterator for IntoIter<T> {
368         fn next_back(&mut self) -> Option<T> {
369             self.iter.next_back()
370         }
371     }
372
373     impl<T> Drop for IntoIter<T> {
```

```
374     fn drop(&mut self) {
375         for _ in &mut *self {}
376     }
377 }
378
379 pub struct Drain<'a, T: 'a> {
380     vector: PhantomData<&'a mut Vector<T>>,
381     iter: RawValIter<T>,
382 }
383
384 impl<'a, T> Iterator for Drain<'a, T> {
385     type Item = T;
386     fn next(&mut self) -> Option<T> {
387         self.iter.next()
388     }
389     fn size_hint(&self) -> (usize, Option<usize>) {
390         self.iter.size_hint()
391     }
392 }
393
394 impl<'a, T> DoubleEndedIterator for Drain<'a, T> {
395     fn next_back(&mut self) -> Option<T> {
396         self.iter.next_back()
397     }
398 }
399
400 impl<'a, T> Drop for Drain<'a, T> {
401     fn drop(&mut self) {
402         // pre-drain the iter
403         for _ in &mut *self {}
404     }
405 }
```

3.2 Строка на основі вектора

Напишемо структуру яка буде реалізовувати строку на основі вектора який состоїть з char.
Код програми:

```rust
use std::{fmt, ops::Add, str::FromStr};

use crate::libs::vector::Vector;

#[derive(Debug, Clone)]
pub struct VecString {
    vec: Vector<char>,
}

impl VecString {
    pub fn new() -> Self {
        Self { vec: Vector::new() }
    }

    pub fn len(&self) -> usize {
        self.vec.len()
    }

    pub fn with_len(len: usize) -> Self {
        Self {
            vec: Vector::with_capacity(len),
        }
    }

    pub fn push(&mut self, c: char) {
        self.vec.push(c);
    }

    pub fn push_str(&mut self, s: &str) {
        for c in s.chars() {
            self.push(c);
        }
    }

    pub fn push_another(&mut self, s: &VecString) {
        for c in s.vec.iter() {
            self.push(c.clone());
        }
    }

    pub fn substring(&self, start: i64, end: i64) -> Self {
        Self {
            vec: self.vec.slice(start, end),
        }
    }

    pub fn full_size(&self) -> usize {
        return self.vec.full_size()
    }
}

impl Add<&str> for VecString {
    type Output = Self;

    fn add(mut self, other: &str) -> Self {
        self.push_str(other);
        self
    }
}

impl Add<&VecString> for VecString {
    type Output = Self;

    fn add(mut self, other: &Self) -> Self {
        self.push_another(&other);
        self
    }
```

```
}

impl From<&str> for VecString {
    fn from(value: &str) -> Self {
        let mut string = Self::with_len(value.len());
        string.push_str(value);
        string
    }
}

#[derive(Debug, PartialEq, Eq)]
pub struct ParseVecStringError;

impl FromStr for VecString {
    type Err = ParseVecStringError;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        Ok(VecString::from(s))
    }
}

impl fmt::Display for VecString {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        for c in self.vec.iter() {
            write!(f, "{c}")?;
        }

        Ok(())
    }
}
```

3.3 Блочно-зв’язне подання строки

Напишемо Блочно-зв’язну строку яка буде реалізована на основі звязного списка, а елементами буде строка яку ми описали вище. Також додоамо два способи для розбиття блока:

1. Блоку певний розмір,і при перевищені якого буде створюватися новий блок

2. Блоки розділені певним символом, при попаданні на такий символ буде створено новий блок, а сам символ не буде записаний

Код програми:

```
use std::fmt;
use std::mem;
use std::ops::Add;
use std::str::FromStr;

use crate::libs::list::double_linked_list::Node;

use super::super::list::double_linked_list::List;
use super::super::utils::clamp_range;
use super::vec::VecString;

#[derive(Clone)]
enum Split {
    Sized(usize),
    Symbol(char),
}

pub struct BlockString {
    list: List<VecString>,
    split: Split,
    len: usize,
    current_block: usize,
}

impl BlockString {
    pub fn new_with_symbol_block(split_char: char) -> Self {
        Self::new_raw(Split::Symbol(split_char))
    }

    pub fn new_with_sized_block(size: usize) -> Self {
        Self::new_raw(Split::Sized(size))
    }

    fn new_raw(split: Split) -> Self {
        let mut list = List::new();
        list.push(VecString::new());
        Self {
            list,
            split,
            len: 0,
            current_block: 0,
        }
    }

    pub fn push(&mut self, c: char) {
        match self.split {
            Split::Sized(size) => {
                let block = self.list.get_mut(self.current_block as i64).unwrap();
                if block.len() + 1 > size {
                    let mut v = VecString::new();
                    v.push(c);
                    self.list.push(v);
                    self.current_block += 1;
                } else {
                    block.push(c);
```

```
                }
            }
            Split::Symbol(symbol) => {
                if symbol == c {
                    self.list.push(VecString::new());
                    self.current_block += 1;
                } else {
                    self.list
                        .get_mut(self.current_block as i64)
                        .unwrap()
                        .push(c);
                }
            }
        };
        self.len += 1;
    }

    pub fn push_block(&mut self, block: VecString) {
        self.len += block.len();
        self.list.push(block);
        self.current_block += 1;
    }

    pub fn push_str(&mut self, s: &str) {
        for c in s.chars() {
            self.push(c);
        }
    }

    pub fn len(&self) -> usize {
        self.len
    }

    pub fn substring(&self, start: i64, end: i64) -> Self {
        let (start, end) = clamp_range(self.len(), start, end);

        let mut res = Self {
            list: List::new(),
            split: self.split.clone(),
            len: 0,
            current_block: 0,
        };

        if start >= end {
            res.list.push(VecString::new());
            return res;
        }

        let is_sized = match self.split {
            Split::Sized(_) => true,
            Split::Symbol(_) => false,
        };
        let mut block_i = 0;
        let mut i = 0;
        let mut blocks_len = 0;
        for total in 0..end {
            let block = self.list.get(block_i).unwrap();
            if i >= block.len() as i64 {
                if total >= start {
                    let start = (start - blocks_len).max(0);
                    res.push_block(block.substring(start, -1));
                }

                block_i += 1;
                if is_sized {
                    blocks_len += i;
                    i = 0;
                } else {
                    blocks_len += i + 1;
                    i = -1;
                };
            }

            i += 1;
        }

```

```
132             if blocks_len <= end {
133                 let block = self.list.get(block_i).unwrap();
134                 let start = (start - blocks_len).max(0);
135                 let end = end - blocks_len - 1;
136                 if !is_sized && end == -1 {
137                     res.push_block(VecString::new());
138                 } else {
139                     res.push_block(block.substring(start, end));
140                 }
141             }
142 
143             res
144         }
145 
146         pub fn full_size(&self) -> usize {
147             let mut size = mem::size_of::<Self>();
148             let len = self.list.len();
149             for i in 0..len {
150                 size += self.list.get(i as i64).unwrap().full_size();
151                 size += mem::size_of::<*mut Node<VecString>>() * 2;
152             }
153             size
154         }
155     }
156 
157     impl Add<&str> for BlockString {
158         type Output = Self;
159 
160         fn add(mut self, other: &str) -> Self {
161             self.push_str(other);
162             self
163         }
164     }
165 
166     impl From<&str> for BlockString {
167         fn from(value: &str) -> Self {
168             let mut string = Self::new_with_sized_block(255);
169             string.push_str(value);
170             string
171         }
172     }
173 
174     #[derive(Debug, PartialEq, Eq)]
175     pub struct ParseBlockStringError;
176 
177     impl FromStr for BlockString {
178         type Err = ParseBlockStringError;
179 
180         fn from_str(s: &str) -> Result<Self, Self::Err> {
181             Ok(BlockString::from(s))
182         }
183     }
184 
185     impl fmt::Display for BlockString {
186         fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
187             let len = self.list.len();
188             for i in 0..len {
189                 self.list.get(i as i64).unwrap().fmt(f)?;
190                 match self.split {
191                     Split::Symbol(c) if i < (len - 1) => write!(f, "{c}")?,
192                     _ => (),
193                 };
194                 // write!(f, "|")?
195             }
196 
197             Ok(())
198         }
199     }
```

3.4 Приклад роботи програми

Код програми для перевірки працездатності всіх строк:

```
use std::time::Instant;
use crate::libs::string::block::BlockString;
use crate::libs::string::vec::VecString;

fn test_all_substrings(test_case: &str) {
    let len = test_case.len();

    let mut s_symbol = BlockString::new_with_symbol_block(' ');
    let mut s_sized = BlockString::new_with_sized_block(4);
    let mut s_vec = VecString::new();
    s_symbol.push_str(test_case);
    s_sized.push_str(test_case);
    s_vec.push_str(test_case);

    for i in 1..=(len + 1) as i64 {
        for j in 0..=len as i64 {
            let s = s_symbol.substring(j, -i);
            let t1 = format!("({j} {}): |{s}|", -i);
            let s = s_sized.substring(j, -i);
            let t2 = format!("({j} {}): |{s}|", -i);
            let s = s_vec.substring(j, -i);
            let t3 = format!("({j} {}): |{s}|", -i);

            println!("{t1}");
            assert_eq!(t1, t2);
            assert_eq!(t2, t3);
        }
    }
}
macro_rules! timeit_substring {
    ($string:ident, $string_src:ident, $title:expr) => {
        println!("{:=^40}", $title);
        let s = 4;
        let e = -5;
        let n = 100;

        $string.push_str($string_src);
        let substring = $string.substring(s, e);

        println!("original: |{}|", $string);
        println!("substring: |{}|", substring);
        println!("full size: {}", $string.full_size());

        let now = Instant::now();
        for _ in 0..n {
            $string.substring(s, e);
        }
        println!("PC time: {} ns  repeats: {n}", now.elapsed().as_nanos());
        println!("{:=^40}\n", "");
    };
}

pub fn main() {
    let test_case = "  01 23 45 67 89  ";

    let mut s_symbol = BlockString::new_with_symbol_block(' ');
    let mut s_sized = BlockString::new_with_sized_block(4);
    let mut s_vec = VecString::new();

    println!("\n");
    timeit_substring!(s_symbol, test_case, "BlockString(Symbol)");
    timeit_substring!(s_sized, test_case, "BlockString(Sized)");
    timeit_substring!(s_vec, test_case, "VecString");
    // test_all_substrings(test_case);
}
```

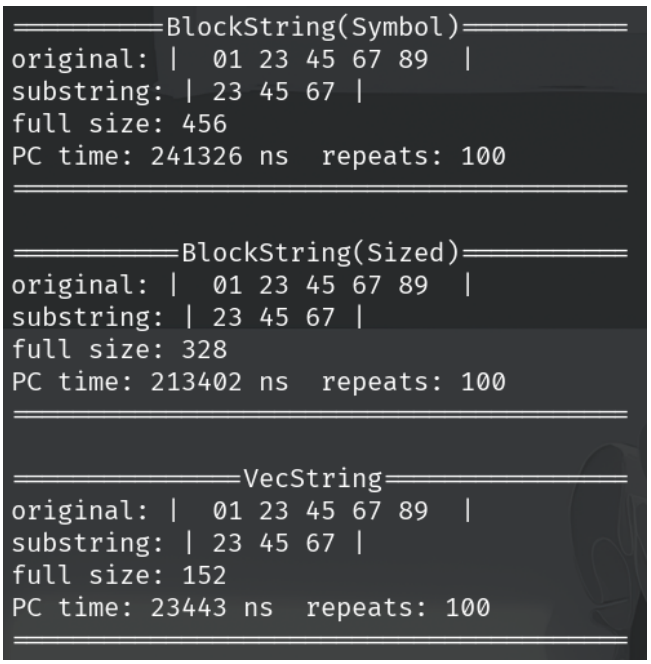

Рис. 1. Приклад роботи

4 Висновки

В ході виконання лабораторної робити було створено 3 варіанти подання строк. Векторне подання строки є саме ефективне за пам’яттю та за часом виконання операції для подання незмінюваних строк. В порівнянні векторне подання в 10 разів швидше справляється з тим щоб зробити під строку. Але блочно-зв’язне подання є також гарним рішенням для дуже великих строк які потребують дуже частих змін, бо в такому випадку ми можемо вирізати та додавати цілі блоки та не перевиділяти память для змін в рядку.