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 №7

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

Набути практичного досвіду та закріпити знання про подання стека, дека, пріоритетної черги та дисципліни їх обслуговування.

Теми для попередньої роботи:

- масиви та списки;
- безпріоритетні та пріоритетні черги;
- дисципліни обслуговування черг.

2 Завдання

Розробити функції, що забезпечують запис та читання запитів із пріоритетної черги, стека або дека. В кожному завданні для організації вказаної черги використати дві структури. Перевірити працездатність розроблених функцій. Послідовність виконання операцій запису та читання обирати випадково. Порівняти результати роботи, зробити висновки.

Завдання за варіантом (5)
Стек. Стек організований на двоспрямованому списку та на масиві і «зростає» від меншої адреси пам’яті до більшої.

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 Linked List

Для цього завдання напишемо власну реалізацю двозвязного списку яка буде використовуватися далі

```rust
use std::{
    fmt::{self, Display},
    ptr,
};

pub struct Node<T> {
    pub value: T,
    pub prev: *mut Node<T>,
    pub next: *mut Node<T>,
}

pub struct List<T> {
    pub head: *mut Node<T>,
    pub tail: *mut Node<T>,
    len: usize,
}

impl<T> List<T> {
    pub fn new() -> Self {
        Self {
            head: ptr::null_mut(),
            tail: ptr::null_mut(),
            len: 0,
        }
    }

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

    pub fn is_empty(&self) -> bool {
        self.len == 0
    }

    pub fn push_front(&mut self, value: T) {
        let new_node = Box::into_raw(Box::new(Node {
            value,
            prev: ptr::null_mut(),
            next: self.head,
        }));

        unsafe {
            if !self.head.is_null() {
                (*self.head).prev = new_node;
            }

            self.head = new_node;

            if self.tail.is_null() {
                self.tail = new_node;
            }

            self.len += 1;
        }
    }

    pub fn push(&mut self, value: T) {
        let new_node = Box::into_raw(Box::new(Node {
            value,
            prev: self.tail,
            next: ptr::null_mut(),
        }));

        unsafe {
            if !self.tail.is_null() {
                (*self.tail).next = new_node;
            }

            self.tail = new_node;
```

```
70
71              if self.head.is_null()      {
72                  self.head   = new_node;
73              }
74
75              self.len    += 1;
76          }
77      }
78
79      pub fn pop(&mut self) -> Option<T> {
80          if self.tail.is_null()      {
81              return None;
82          }
83
84          unsafe {
85              let old_tail    = self.tail;
86              self.tail     = (*old_tail).prev;
87              (*old_tail).prev    = ptr::null_mut();
88
89              if self.tail.is_null()      {
90                  self.head   = ptr::null_mut();
91              } else {
92                  (*self.tail).next       = ptr::null_mut();
93              }
94
95              self.len    -= 1;
96              Some(Box::from_raw(old_tail).value)
97          }
98      }
99
100     pub fn remove(&mut self, mut index: i64) -> Option<T> {
101         if index < 0 {
102             index = self.len    as i64 + index;
103         }
104
105         if index >= self.len    as i64 {
106             return None;
107         }
108
109         unsafe {
110             let mut node = self.head;
111             for _ in 0..index   {
112                 node = (*node).next;
113             }
114
115             if !(*node).next.is_null()      {
116                 (*(*node).next).prev    = (*node).prev
117             }
118
119             if !(*node).prev.is_null()      {
120                 (*(*node).prev).next    = (*node).next
121             }
122
123             if node == self.head    {
124                 self.head   = (*node).next;
125             }
126
127             if node == self.tail    {
128                 self.tail     = (*node).prev;
129             }
130
131             self.len    -= 1;
132             Some(Box::from_raw(node).value)
133         }
134     }
135
136     pub fn get(&self,   mut index: i64) -> Option<&T> {
137         if index < 0 {
138             index = self.len    as i64 + index;
139         }
140
141         if index >= self.len    as i64 {
142             return None;
143         }
144
145         unsafe {
```

```
146                 let mut node = self.head;
147                 for _ in 0..index {
148                     node = (*node).next;
149                 }
150                 return Some(&mut (*node).value);
151             }
152         }
153
154         pub fn get_mut(&mut self, mut index: i64) -> Option<&mut T> {
155             if index < 0 {
156                 index = self.len as i64 + index;
157             }
158
159             if index >= self.len as i64 {
160                 return None;
161             }
162
163             unsafe {
164                 let mut node = self.head;
165                 for _ in 0..index {
166                     node = (*node).next;
167                 }
168                 return Some(&mut (*node).value);
169             }
170         }
171     }
172
173     impl<T: PartialEq> List<T> {
174         pub fn index(&mut self, value: T) -> Option<usize> {
175             let mut node = self.head;
176
177             unsafe {
178                 for i in 0..self.len {
179                     if (*node).value == value {
180                         return Some(i);
181                     }
182                     node = (*node).next;
183                 }
184             }
185
186             None
187         }
188     }
189
190     impl<T: Display> Display for List<T> {
191         fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
192             unsafe {
193                 let mut node = self.head;
194                 for _ in 0..self.len {
195                     write!(f, "{} ", (*node).value)?;
196                     node = (*node).next;
197                 }
198             }
199             Ok(())
200         }
201     }
202
203     impl<T> Drop for List<T> {
204         fn drop(&mut self) {
205             while self.pop().is_some() {}
206         }
207     }
```

3.3 Спільний інтерфейс для стека

Для подальших імплементацій стека напишемо спільний інтерфейс

```
1  pub trait StackMethods<T> {
2      fn pop(&mut self) -> Option<T>;
3      fn push(&mut self, value: T);
4  }
```

3.4 Стек на основі вектора

```
use super::super::vector::Vector;
use super::StackMethods;

pub struct Stack<T> {
    container:  Vector<T>,
}

impl<T> Stack<T> {
    pub fn new() -> Stack<T> {
        Stack {
            container:  Vector::<T>::new(),
        }
    }
}

impl<T> StackMethods<T> for Stack<T> {
    fn pop(&mut self) -> Option<T> {
        self.container.pop()
    }

    fn push(&mut self, value: T) {
        self.container.push(value);
    }
}
```

3.5 Стек на основі списка

```rust
use std::fmt::{self, Display};

use super::super::list::double_linked_list::List;
use super::StackMethods;

pub struct Stack<T> {
    container: List<T>,
}

impl<T> Stack<T> {
    pub fn new() -> Stack<T> {
        Stack {
            container: List::<T>::new(),
        }
    }
}

impl<T> StackMethods<T> for Stack<T> {
    fn pop(&mut self) -> Option<T> {
        self.container.pop()
    }

    fn push(&mut self, value: T) {
        self.container.push(value);
    }
}

impl<T: Display> Display for Stack<T> {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        self.container.fmt(f)
    }
}
```

3.6 Приклад роботи програми

Напишемо програму яка буде перевіряти та порівнювати скільки часу займає додавання та видалення елементів зі стека для обох варіантів Код програми для перевірки:

```rust
use std::time::Instant;

use crate::libs::stack::list_stack::Stack as ListStack;
use crate::libs::stack::vector_stack::Stack as VectorStack;
use crate::libs::stack::StackMethods;

pub fn perf_test_stack<T: StackMethods<i32>>(mut stack: T) -> (u128, u128) {
    let now = Instant::now();
    for _ in 0..10000 {
        stack.push(1);
    }
    let push = now.elapsed().as_nanos();

    let now = Instant::now();
    for _ in 0..10000 {
        stack.pop();
    }
    let pop = now.elapsed().as_nanos();

    return (push, pop);
}

pub fn main() {
    let time_list = perf_test_stack(ListStack::new());
    let time_vector = perf_test_stack(VectorStack::new());

    println!(
        "List Stack Times (Push/Pop): {} ns / {} ns",
        time_list.0, time_list.1
    );
    println!(
        "Vector Stack Times (Push/Pop): {} ns / {} ns",
        time_vector.0, time_vector.1
    );
}
```

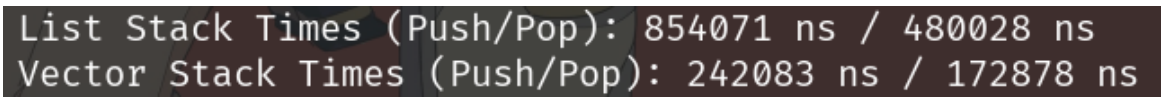

Рис. 1. Приклад роботи

4 Висновки

В ході виконання лабораторної робити було створено 2 варіанти подання стека. За результатами порівняння було виявлено, що стек на основі вектора є ефективнішим як за доданням, так і за видаленням. Тому використання такого подання буде ефективнішим для більшості завдань. Проте використання стека на основі списика може бути раціональним коли нам треба зберігати великий обсяг даних в елементах стека, бо не треба виділяти неперервну ділянку пам’яті для цих елементів на відміну від векторного представлення.