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 №8

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 навичок та закріпити знання при виконанні операцій на мультисписках та нелінійних списках.

Теми для попередньої роботи:

- набори даних – списки: лінійні, кільцеві, мультисписки, нелінійні списки;
- операції на списках.

2 Завдання

Для варіантів завдань 1 – 13 розробити програму, що створює список списків (нелінійний список). Передбачити такі функції:

- додавання елементів у список та підсписок (при додаванні елемента у головний список додається і відповідний підсписок);
- видалення елементів зі списку та підсписків (при видаленні елемента з головного списку видаляється і пов’язаний з ним підсписок);
- видача вмісту списків та підсписків у консоль;
- видалення списків.

Завдання обрати у табл. 8.1 згідно зі своїм номером у журналі групи.

З/П	Список	Підсписок
5	Маршрути автобусів	Назви пунктів зупинок автобуса

Рис. 1. Завдання за варіантом (5)

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 Non linear list

Зробимо його елемент таким, що в ньому можуть лежати такий же non linear list та якийсь значення. Таким чином ми можемо побудувати структуру будь-якої ієрархії та при цьому зберігати значення. Для взаємодії з списком ми реалізуємо такі функції:

- Додання елемента. Якщо такої вкладеності ще нема, то вона автоматично створиться.
- Додання підсписку. Якщо такої вкладеності ще нема, то вона автоматично створиться.
- Видалення елемента / підсписка.
- Взяття елемента / підсписка.
- Вивід на екран.

```
use std::fmt::Debug;
use std::{fmt, mem, vec};
use std::{fmt::Display, ptr};

use super::double_linked_list::List;

#[derive(Debug)]
pub struct Node<T>(Option<T>, *mut List<Node<T>>);

#[derive(Debug)]
pub struct MultiList<T> {
    root: Node<T>,
    dimentions: Option<usize>,
    limits: Vec<Option<i64>>,
}

impl<T> MultiList<T> {
    pub fn new(
        dimentions: Option<usize>,
        limits: Option<Vec<Option<i64>>>,
    ) -> Result<MultiList<T>, String> {
        let (dimentions, limits) = match (dimentions, limits) {
            (None, None) => (None, Vec::new()),
            (None, Some(limits)) => (None, limits),
            (Some(dimentions), None) => {
                Self::validate_dimantions(dimentions, 0)?;
                (None, Vec::new())
            }
            (Some(dimentions), Some(limits)) => {
                Self::validate_dimantions(dimentions, limits.len())?;
                (Some(dimentions), limits)
            }
        };

        Ok(Self {
            root: Self::new_list_node(),
            dimentions,
            limits,
        })
    }

    fn validate_dimantions(dimentions: usize, limits: usize) -> Result<(), String> {
        if dimentions == 0 {
            return Err("Dimentions must be greater than zero".into());
        }
```

```
46
47          if limits > dimentions {
48              return Err("Dimentions must be less than or equal to the limits vector length".into());
49          }
50
51          Ok(())
52      }
53
54      fn new_list_node() -> Node<T> {
55          Node(None, Box::into_raw(Box::new(List::<Node<T>>::new())))
56      }
57
58      fn new_value_node(value: T) -> Node<T> {
59          Node(Some(value), ptr::null_mut())
60      }
61
62      fn new_void_node() -> Node<T> {
63          Node(None, ptr::null_mut())
64      }
65
66      fn check_limits(&self, index: i64, level: usize) -> bool {
67          if level >= self.limits.len() {
68              return true;
69          }
70
71          match self.limits[level] {
72              None => false,
73              Some(limit) => index < limit,
74          }
75      }
76  }
77
78  impl<'a, T> MultiList<T> {
79      fn _get_node(&self, node: &'a Node<T>, indices: Vec<i64>, level: usize) -> Option<&'a Node<T>> {
80          // println!("level: {}", level);
81          if level >= indices.len() {
82              return Some(node);
83          }
84
85          if !self.check_limits(indices[level], level) {
86              return None;
87          }
88
89          unsafe {
90              if node.1.is_null() {
91                  return None;
92              }
93
94              let tmp = (*node.1).get_mut(indices[level] as i64);
95              if tmp.is_none() {
96                  return None;
97              }
98              self._get_node(tmp.unwrap(), indices, level + 1)
99          }
100     }
101
102     fn get_node(&self, indices: Vec<i64>) -> Option<&Node<T>> {
103         self._get_node(&self.root, indices, 0)
104     }
105
106     pub fn get_value(&self, indices: Vec<i64>) -> Option<&T> {
107         match self.get_node(indices) {
108             None => None,
109             Some(node) => match &node.0 {
110                 None => None,
111                 Some(value) => Some(value),
112             },
113         }
114     }
115
116     pub fn get_list(&self, indices: Vec<i64>) -> Option<&List<Node<T>>> {
117         match self.get_node(indices) {
118             None => None,
119             Some(node) => {
120                 if node.1.is_null() {
121                     return None;
```

```
122                     }
123                     unsafe { Some(&(*node.1)) }
124                 }
125             }
126         }
127     }
128
129     impl<T> MultiList<T> {
130         fn _insert_node(
131             &mut self,
132             parent: &mut Node<T>,
133             indices: Vec<i64>,
134             node: Node<T>,
135             level: usize,
136             insert_value: bool,
137             insert_list: bool,
138         ) {
139             // println!("level {}", level);
140             if level >= indices.len() {
141                 // println!("{:?}", indices);
142                 // println!(">)");
143                 if insert_value {
144                     parent.0 = node.0;
145                 }
146                 if insert_list {
147                     parent.1 = node.1;
148                 }
149                 return;
150             }
151
152             let index = indices[level];
153             if !self.check_limits(index, level) {
154                 return;
155             }
156
157             unsafe {
158                 if parent.1.is_null() {
159                     parent.1 = Box::into_raw(Box::new(List::<Node<T>>::new()))
160                 }
161
162                 let list = parent.1;
163                 if list.is_null() {
164                     return;
165                 }
166
167                 // println!("dopovn");
168                 for _ in 0..=(index - (*list).len() as i64) {
169                     (*list).push(Self::new_void_node());
170                 }
171
172                 let next = (*list).get_mut(index);
173                 if next.is_none() {
174                     return;
175                 }
176
177                 self._insert_node(
178                     next.unwrap(),
179                     indices,
180                     node,
181                     level + 1,
182                     insert_value,
183                     insert_list,
184                 )
185             }
186         }
187
188         fn insert_node(
189             &mut self,
190             indices: Vec<i64>,
191             node: Node<T>,
192             insert_value: bool,
193             insert_list: bool,
194         ) {
195             let mut root = mem::replace(&mut self.root, Self::new_list_node());
196             let res = self._insert_node(&mut root, indices, node, 0, insert_value, insert_list);
197             self.root = root;
```

```
        res
    }

    pub fn insert_value(&mut self, indices: Vec<i64>, value: T) {
        self.insert_node(indices, Self::new_value_node(value), true, false);
    }

    pub fn insert_list(&mut self, indices: Vec<i64>) {
        self.insert_node(indices, Self::new_list_node(), false, true);
    }

    pub fn delete_value(&mut self, indices: Vec<i64>) {
        self.insert_node(indices, Self::new_void_node(), true, false);
    }

    pub fn delete_list(&mut self, indices: Vec<i64>) {
        self.insert_node(indices, Self::new_void_node(), false, true);
    }
}

macro_rules! display_node {
    ($f:ident, $fmt_none:expr, $fmt_value:expr, $fmt_list:expr, $fmt_list_and_value:expr, $node:expr) => {
        match $node {
            Some(item) => match item {
                Node(None, list) => {
                    if list.is_null() {
                        write!($f, $fmt_none)?;
                    } else {
                        write!($f, $fmt_list)?;
                    }
                }
                Node(Some(value), list) => {
                    if list.is_null() {
                        write!($f, $fmt_value, value)?;
                    } else {
                        write!($f, $fmt_list_and_value, value)?;
                    }
                }
                _ => write!($f, $fmt_none)?,
            },
            None => write!($f, $fmt_none)?,
        };
    };
}

impl<T: Display> MultiList<T> {
    fn display_one_list(
        &self,
        f: &mut fmt::Formatter<'_>,
        node: &Node<T>,
        indices: &Vec<i64>,
    ) -> fmt::Result {
        write!(f, "[")?;
        for i in 0..(indices.len() as i64 - 1) {
            write!(f, "{},␣", indices[i as usize])?;
        }
        if indices.len() > 0 {
            write!(f, "{}", indices[indices.len() - 1])?;
        }
        write!(f, "]:␣")?;

        let list = unsafe { &mut (*node.1) };
        for i in 0..(list.len() as i64 - 1) {
            display_node!(f, "None,␣", "{},␣", "[],␣", "{}+[],␣", list.get_mut(i));
        }

        if list.len() > 0 {
            display_node!(
                f,
                "None",
                "{}",
                "[]",
                "{}+[]",
                list.get_mut(list.len() as i64 - 1)
            );
        }
```

```
274
275             writeln!(f)?;
276
277             Ok(())
278         }
279
280         fn display(
281             &self,
282             f: &mut fmt::Formatter<'_>,
283             parent: &Node<T>,
284             indices: &Vec<i64>,
285             level: usize,
286         ) -> fmt::Result {
287             if parent.1.is_null() {
288                 return Ok(());
289             }
290
291             self.display_one_list(f, parent, indices)?;
292
293             let list = unsafe { &mut (*parent.1) };
294
295             for i in 0..list.len() {
296                 let node = list.get_mut(i as i64);
297                 if node.is_none() {
298                     continue;
299                 }
300                 let mut new_indices = indices.to_vec();
301                 new_indices.push(i as i64);
302                 self.display(f, node.unwrap(), &new_indices, level + 1)?;
303             }
304             Ok(())
305         }
306     }
307
308     impl<T: Display + Debug> Display for MultiList<T> {
309         fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
310             self.display(f, &self.root, &vec![], 0)?;
311             Ok(())
312         }
313     }
```

3.2 Приклад роботи програми

Для перевірки працездатності напишемо програму яка буде автоматично генерувати значення для списків. Потім виконаємо наступні дії та виведемо обидві версії на екран для порівняння, а також виділемо зміни елементи:

1. Додамо два елементи у за такими індексами [2, 2, 2] та потім [2, 2, 0]. Таким чином перевіримо автоматичне додання підсписка та додання елементів.
2. Видалимо список за індексом [6]
3. Видалимо значення за індексом [7]
4. Видалимо значення та список за індексом [8]

Код програми для перевірки:

```
use crate::libs::list::non_linear_list::MultiList;

fn present() {
    let mut list = match MultiList::<i64>::new(None, None) {
        Ok(list) => list,
        Err(_) => {
            panic!("Cannot␣create␣list")
        }
    };

    for i in 0..10 {
        list.insert_value(vec![i], i);
        for j in 0..=i {
            list.insert_value(vec![i, j], j);
            for k in 0..=j {
                list.insert_value(vec![i, j, k], k);
            }
        }
    }

    println!("{}", list);
}

fn lab() {
    let mut list = match MultiList::<String>::new(None, None) {
        Ok(list) => list,
        Err(_) => {
            panic!("Cannot␣create␣list")
        }
    };

    for i in 0..10 {
        let station = format!("Path␣{i}");
        list.insert_value(vec![i], station);
        for j in 0..=i {
            let parh = format!("Station␣{j}");
            list.insert_value(vec![i, j], parh);
        }
    }

    for i in 0..10 {
        let elem = list.get_value(vec![i]);
        print!("{:?}␣", elem)
    }
    println!("\n");

    for i in 0..10 {
        let elem = list.get_value(vec![4, i]);
```

```
49          print!("{:?}␣", elem)
50      }
51      println!("\n");
52
53      println!("{}", list);
54      list.insert_value(vec![2, 2, 2], "description␣2".into());
55      list.insert_value(vec![2, 2, 0], "description␣1".into());
56      list.delete_value(vec![2, 1]);
57      list.delete_list(vec![6]);
58      list.delete_value(vec![7]);
59      list.delete_list(vec![8]);
60      list.delete_value(vec![8]);
61      println!("{}", list);
62  }
63
64  pub fn main() {
65      // present();
66      lab();
67  }
```

```
Some("Path 0") Some("Path 1") Some("Path 2") Some("Path 3") Some("Path 4") Some("Path 5") Some("Path 6") Some("Path 7") Some("Path 8") Some("Path 9")

Some("Station 0") Some("Station 1") Some("Station 2") Some("Station 3") Some("Station 4") None None None None None

[]: Path 0+[], Path 1+[], Path 2+[], Path 3+[], Path 4+[], Path 5+[], Path 6+[], Path 7+[], Path 8+[], Path 9+[]
[0]: Station 0
[1]: Station 0, Station 1
[2]: Station 0, Station 1, Station 2
[3]: Station 0, Station 1, Station 2, Station 3
[4]: Station 0, Station 1, Station 2, Station 3, Station 4
[5]: Station 0, Station 1, Station 2, Station 3, Station 4, Station 5
[6]: Station 0, Station 1, Station 2, Station 3, Station 4, Station 5, Station 6
[7]: Station 0, Station 1, Station 2, Station 3, Station 4, Station 5, Station 6, Station 7
[8]: Station 0, Station 1, Station 2, Station 3, Station 4, Station 5, Station 6, Station 7, Station 8
[9]: Station 0, Station 1, Station 2, Station 3, Station 4, Station 5, Station 6, Station 7, Station 8, Station 9

[]: Path 0+[], Path 1+[], Path 2+[], Path 3+[], Path 4+[], Path 5+[], Path 6, [], None, Path 9+[]
[0]: Station 0
[1]: Station 0, Station 1
[2]: Station 0, None, Station 2+[]
[2, 2]: description 1, None, description 2
[3]: Station 0, Station 1, Station 2, Station 3
[4]: Station 0, Station 1, Station 2, Station 3, Station 4
[5]: Station 0, Station 1, Station 2, Station 3, Station 4, Station 5
[7]: Station 0, Station 1, Station 2, Station 3, Station 4, Station 5, Station 6, Station 7
[9]: Station 0, Station 1, Station 2, Station 3, Station 4, Station 5, Station 6, Station 7, Station 8, Station 9
```

Рис. 2. Приклад роботи

4 Висновки

В ході виконання лабораторної робити було створено нелінійний список на базі двозв’язного списку.