ساختمان داده‌ها (۲۲۸۲۲)

مدرس: حسین بومری

[زمستان ۹۹]

جلسه ۱۰: پیاده‌سازی لیست پیوندی دوطرفه و مرتب سازی بازگشتی و غیربازگشتی نگارنده: سهیل کشاورز

در این جلسه به پیاده‌سازی مباحث مطرح شده در جلسات قبل،در محیط جاوا پرداخته شده‌است. نخست لیست پیوندی دوطرفه [۱] ساخته‌شد که یک پایه برای صف [۲] و پشته [۳] می‌باشد. در قسمت دوم کلاس هم مرتب‌سازی ادغامی [۴] و انتخابی [۵] نوشته شد.

۱ لیست پیوندی دوطرفه

با توجه به ساده بودن پیاده‌سازی لیست یک‌طرفه، پیاده‌سازی‌ها با لیست دوطرفه شروع شد. توجه داریم که در لیست یک‌طرفه بسیاری از جزییات لیست دوطرفه که در ادامه خواهندآمد نادیده گرفته می‌شود بنابراین برای پیاده‌سازی آن، از مباحث پیش‌رو می‌توان استفاده کرد. برای پیاده‌سازی لیست دوطرفه با توجه به ساختار جاوا و نیازهایی که می‌توان برای این داده‌ساختار تعریف کرد، کد زیر توسعه داده می‌شود. در ادامه، اجزای این کد توضیح داده خواهندشد. دقت داریم که همچون ArrayList ،انواع داده‌ها میتوانند در لیست قرار بگیرد بنابر این از جنریک (نوع‌داده با علامت T) استفاده می‌شود.

```
public class DoublyLinkedList<T> {
    static class Node<T> {...}
    Node<T> start;
    Node<T> end;
    int size;
    public DoublyLinkedList() {...}
    boolean isEmpty() {...}
    void pushFront(T value) {...}
    T popFront() {...}
    T getFront() {...}
    void pushBack(T value) {...}
    T popBack() {...}
    T getBack() {...}
    T get(int index) {...}
    void insert(T value, int index) {...}
    T remove(int index) {...}
    ArrayList<T> toArrayList() {...}
    void print() {...}
}
```

[1] Doubly Linked List
[2] Queue
[3] Stack
[4] Merge Sort
[5] Selection Sort

class Node

نخست نیاز است تا نود [۶] را در یک کلاس تعریف کنیم. کلاس داخلی [۷] استاتیک نود در زیر تعریف شده است. استاتیک تعریف شدن برای این است که در عمل ما باید قادر باشیم تا بدون وجود لیست هم نود بسازیم (نود بدون لیست معنی‌دار است) و داخلی بودن آن برای این است که مشخص شود که این نود مخصوص لیست دوطرفه است؛ به همین جهت اگر یک لیست یک‌طرفه هم بسازیم برای آن یک نود مخصوص به خود را تعریف می‌کنیم.

```
static class Node<T> {
1   T value;
2   Node<T> next;
3   Node<T> previous;
4   public Node(T value, Node<T> previous, Node<T> next) {
5         this.value=value;
6         this.next=next;
7         this.previous=previous; }
8   public Node(T value) {
9         this(value,null,null); }
10  public Node() {
11        this(null); }
    }
```

از قبل به یاد داریم که یک نود در لیست پیوندی دوطرفه، باید یک مقدار (خط ۱)داشته باشد و به دو نود قبل و بعدش اشاره (خط ۲ و ۳) کند.

constructor and fields

مشخصه‌های یک لیست دوطرفه و پیوندی، دو اشاره گر به اول [۸] و آخر [۹] لیست و مقدار اندازه لیست می‌باشد. از سازنده هم انتظار داریم تا یک لیست پیوندی خالی تولید کند تا با توابعی که به آن‌ها خواهیم پرداخت، اعضای آن را مشخص کنیم.

```
public DoublyLinkedList() {
1         start=null;
2         end=null;
3         size=0;
    }
```

همچنین یک متد ابتدایی مهم در داده‌ساختارها، isEmpty است.

```
boolean isEmpty() {
1   return size == 0;
    }
```

[6] Node
[7] Nested
[8] start / front
[9] Stack / back

Front

متد زیر مستقیما یک مقدار را به ابتدای لیست اضافه می‌کند (به عبارتی دیگر نود آغازین جدیدی خواهیم داشت که برای نود آغازین قبلی، نود پیشین محسوب می‌شود). باید حواسمان باشد که در یک حالت خاص، ممکن است به یک لیست خالی عضو اضافه کنیم که در آن حالت اشاره گر به آغاز لیست، null است.

```
void pushFront(T value) {
1   Node<T> newNode = new Node<>(value,null,start);
2   if (isEmpty()) {
3         end=newNode;}
4   else {
5         start.previous=newNode; }
6   start=newNode;
7   size++;
    }
```

برای دیدن عنصر ابتدایی:

```
T getFront() {
1   if (isEmpty()) {
2         throw new IndexOutOfBoundsException("List is empty"); }
3   return start.value;
    }
```

برای حذف عنصر اول هم به خالی بودن یا نبودن لیست پس از حذف، و وضعیت اشاره‌گرهای نود ها و لیست بدون عنصر اول توجه داریم و آن را به شکل زیر پیاده می‌کنیم:

```
T popFront() {
1   T result =getFront();
2   start = start.next;
3   size − −;
4   if (isEmpty()) {
5         end = null; }
6   else {
7         start.previous =null; }
8   return result;
    }
```

Back

با توجه به اینکه در لیست دوطرفه به آخر هم اشاره گر داریم، مشابه کار هایی که در اول لیست کردیم را برای آخر لیست هم انجام می‌دهیم. برای اینکار جزییات متد های قبلی را عکس می‌کنیم.

```
void pushBack(T value){
1   Node<T> newNode = new Node<>(value,end,null);
2   if (isEmpty()) {
3         start=newNode; }
4   else {
5         end.next=newNode; }
6   end=newNode;
7   size++;
    }

T popBack() {
1   T result = getBack();
2   end=end.previous;
3   size- -;
4   if (isEmpty()) {
5         start=null; }
6   else {
7         end.next=null; }
8   return result;
    }

T getBack() {
1   if (isEmpty()) {
2         throw new IndexOutOfBoundsException("List is empty"); }
3   return end.value;
    }
```

get

چنانچه بخواهیم مقدار iام یک لیست را ببینیم از متد زیر استفاده می‌شود. همانند آنچه برای تابع toArrayList گفته خواهد شد، اینجا حلقه (خط ۴) را با while نیز می‌توان پیاده کرد.

```
T get(int index) {
1   if (index>=size) {
2         throw new IndexOutOfBoundsException(); }
3   Node<T> current = start;
4   for (int i=0; i<index; i++, current=current.next);
5   return current.value;
    }
```

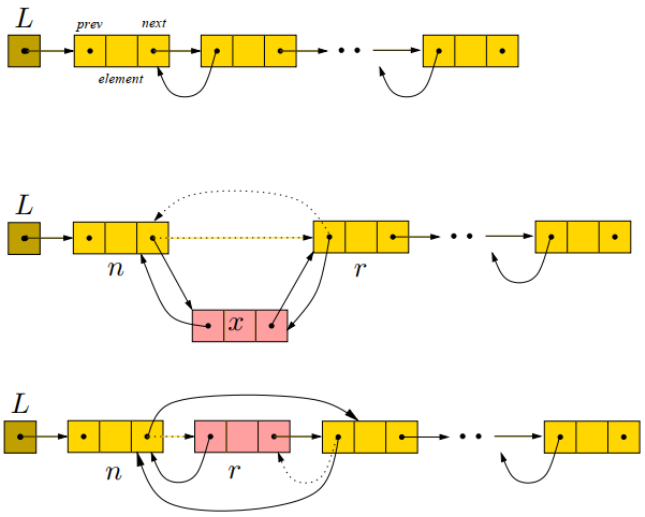
Insert

متدی هم برای اضافه کردن یک عنصر در یک اندیس خاص می‌نویسیم. برای پیاده‌سازی آن current به عنصر جلو اشاره می‌کند و previous به عنصر قبل و ما میخواهیم بین این دو نود جدید را اضافه کنیم. در حین پیاده سازی باید به نیازهای مربوط به زمان خالی بودن لیست یا اضافه کردن به ابتدا (اندیس ۰) و انتها (اندیس سایز) توجه داشت. البته اضافه کردن به ابتدا و انتها قبل در بالاتر هم پیاده‌شده بودند.

```
void insert(T value, int index) {
 1  if (index>size) {
 2        throw new IndexOutOfBoundsException(); }
 3  Node<T> current = start;
 4  Node<T> previous = null;
 5  for (int i=0; i<index; i++) {
 6        previous=current;
 7        current=current.next; }
 8  Node<T> newNode = new Node<>(value, previous, current);
 9  if (previous!=null) {
10        previous.next=newNode; }
11  else {                  // insertion in back (index = size)
12        start=newNode; }
13  if (current!=null) {
14        current.previous = newNode;}
15  else {                  // insertion in front (index=0)
16        end=newNode; }
17  size++;
    }
```

Remove

برای حذف کردن یک عنصر با اندیس i، مطابق با شکل ۱، کاری می‌کنیم که نه از ابتدا بتوان به عنصر مورد نظر رسید(خطوط ۶ تا ۹)، و نه از انتها(خطوط ۱۰ تا ۱۳). برای مثال در شکل مشاهده می‌شود که برای حذف نود صورتی رنگ، نود های عقب و جلوی آن را باید به هم متصل کنیم (اشاره گر نقطه‌چین در شکل وسط). در پایین شکل هم حالت نهایی مشاهده می‌شود که در آن صورت دیگر به نود صورتی دسترسی نداریم.



شکل ۱: نحوه‌ی حذف یک نود.(L را اشاره گر به انتها در نظر بگیرید). [۱]

```
T remove(int index) {
 1  if (index>=size) {
 2        throw new IndexOutOfBoundsException(); }
 3  Node<T> current = start;
 4  for(int i=0;i<index;i++, current=current.next);
 5  T result = current.value;
 6  if (current!=start) {
 7        current.previous.next = current.next; }
 8  else {
 9        start=current.next; }
10  if (current!=end) {
11        current.next.previous = current.previous; }
12  else {
13        end=current.previous; }
14  size - -;
15  if (isEmpty()) {
16        start=null;
17        end=null; }
18  return result;
    }
```

others

یک متد هم برای ایجاد ArrayList ایجاد می‌کنیم. اگر بخواهیم از این لیست یک آرایه بدست آوریم مطابق مباحث برنامه نویسی پیشرفته، T[] نداریم و باید به کمک رفلکشن (متد Array.newInstance()) آرایه را ایجاد کنیم. بنابراین بدون اینکه لازم باشد از مباحث پیشرفته جاوا استفاده شود، ArrayList می‌سازیم.

```
ArrayList<T> toArrayList() {
1   ArrayList<T> result = new ArrayList<>(size);
2   for (Node<T> current = start;current!=null; current=current.next) {
3         result.add(current.value); }
4   return result;
    }
```

راه دیگر دور زدن به این شکل است که به‌جای حلقه‌ی خط ۲و۳، عبارت زیر را جایگزین کنیم:

```
Node<T> current = start;
while(current!=null){
    //operations
    current=current.next; }
```

یا حتی می‌توان این دو خط را خلاصه تر هم کرد :

```
for (Node<T> current = start; current!=null; result.add(current.value), current=current.next);
```

با توجه به فرمت مورد نظر، یک متد برای پرینت لیست هم تهیه می‌کنیم. فرمت مورد نظر ما در اینجا به‌صورت $[obj_1, obj_2, ..., obj_n]$ می‌باشد.

```
void print() {
1   System.out.print("[");
2   Node<T> current = start;
3   while (current!=null) {
4         System.out.print(current.value);
5         current=current.next;
6         if (current!=null) {
7               System.out.print(", "); } }
8   System.out.println("]");
    }
```

تست برنامه

در نهایت هم برنامه ای می‌نویسیم تا متدها و edge case ها بررسی شوند. پاسخ نهایی این برنامه به‌صورت زیر خواهد بود:

[Soheil, Hossein, Mobin, Sahel, Ali]

DoublyLinkedListMain

```
    public static void main(String[] args) {
1       DoublyLinkedList<String> list = new DoublyLinkedList<>();
2       list.pushFront(”Hossein”);
3       list.pushFront(”Ali”);
4       list.pushBack(”Sahel”);
5       list.pushBack(”Aeiria”);
6       list.insert(”Soheil”, 2);
7       list.insert(”Mobin”, 2);
8       list.print();
9       list.remove(3);
10      list.print();
11      list.popBack();
12      list.print();
13      list.popFront();
14      list.print();
15      list.insert(”Soheil”, 0);
16      list.print();
17      list.insert(”Ali”, list.size);
18      list.print();
        }
```

بدین ترتیب لیست پیوندی دوطرفه نوشته شد. برای تمرین دانشجویان اکنون می‌توانند با توجه به ویژگی‌های صف و پشته، به پیاده‌سازی این دو داده‌ساختار بپردازند.

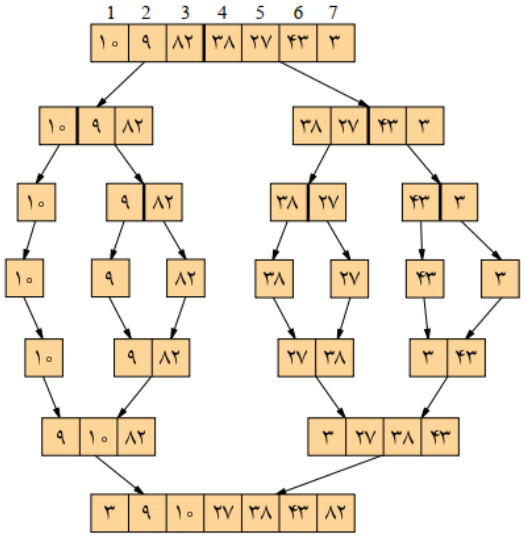
۲ مرتب‌سازی

در این قسمت به با رجوع به توابع بازگشتی و غیربازگشتی مطرح‌شده در جلسات پیشین، توابع مرتب‌سازی در کلاسی به‌نام Sorts در قالب متدهای استاتیک نوشته شدند.

مرتب‌سازی ادغامی

برای این روش مرتب‌سازی ابتدا لیست به دو قسمت تقسیم می‌شود و سپس برای هرقسمت تابع به صورت بازگشتی صدا زده می‌شود. در نهایت هم دو قسمت با هم ادغام می‌شوند. در شکل ۲ یک مثال از این الگوریتم دیده می‌شود. در کد نوشته شده، تقسیم در خطوط ۳ تا ۹، بازگشت در خطوط ۱۰ و ۱۱ و ادغام در خط ۱۲ دیده می‌شوند.

ضمنا می‌خواهیم مرتب سازی ادغامی را به‌صورت inPlace پیاده‌سازی کنیم چرا که بدین صورت حافظه کمتری هم مصرف شود. در خود جاوا هم دیده‌ایم که مرتب‌کردن چیزی برنمیگرداند و خود داده‌ساختار مرتب می‌شود. اگر متد نوشته شده قرار بود لیست برگرداند، باید در خط ۲ به کمک (ArrayList<T>)values.clone() یک کپی از لیست برگشت می‌دادیم(چرا؟).



شکل ۲: مثالی از الگوریتم مرتب‌سازی ادغامی [۱]

```
static <T extends Comparable<T> > void mergeSort(ArrayList<T> values) {
    // Memmory analysis: M(n) = 2 * M(n/2) + n → M(n) = O(nlgn)
1   if (values.size()<2) {
2       return ; }
3   ArrayList<T> left = new ArrayList<>(values.size()/2);
4   ArrayList<T> right = new ArrayList<>((values.size()+1)/2);
5   int i=0;
6   for (; i<values.size()/2; i++) {
7       left.add(values.get(i)); }
8   for (; i<values.size(); i++) {
9       right.add(values.get(i)); }
10  mergeSort(right);
11  mergeSort(left);
12  merge(left, right, values);
    }
```

برای ادغام هم ابتدا دو اشاره‌گر به ابتدای دو لیست ورودی اختصاص می‌دهیم. سپس لیست اصلی خالی می‌شود تا بعدا از عناصر مرتب شده دو لیست دیگر پر شود. در ادامه هم عناصر متناظر با اشاره‌گر ها با هم مقایسه می‌شوند و سپس عنصر کوچکتر وارد لیست اصلی

می‌شود و اشاره‌گرش جلو می‌رود.

```
static <T extends Comparable<T> > void merge(ArrayList<T> left, ArrayList<T> right, ArrayList<T>
result) {
 1        int leftPointer=0;
 2        int rightPointer=0;
 3        result.clear();
 4        while (leftPointer<left.size() && rightPointer<right.size()) {
 5             if (left.get(leftPointer).compareTo(right.get(rightPointer))<0) {
 6                  result.add(left.get(leftPointer));
 7                  leftPointer++; }
 8             else {
 9                  result.add(right.get(rightPointer));
10                  rightPointer++; }}
          // when iteration in 'right' is finished
11        while (leftPointer<left.size()) {
12             result.add(left.get(leftPointer));
13             leftPointer++; }
          // when iteration in 'left' is finished
14        while (rightPointer<right.size()) {
15             result.add(right.get(rightPointer));
16             rightPointer++;}
          }
```

به عنوان نکته، می‌توان خط ۶ و ۷ (یا موارد مشابه در ادامه‌ی کد) را بصورت زیر نوشت که کمی سریع تر هم می‌باشد.

```
result.add(left.get(leftPointer)++)
```

مرتب سازی انتخابی

حال برای مقایسه سرعت الگوریتم‌های مرتب‌سازی، مرتب‌سازی انتخابی هم نوشته می‌شود.

```
1  static <T extends Comparable<T> > void selectionSort(ArrayList<T> values) {
2       for (int i=0; i<values.size(); i++) {
3            int minIndex = i;
4            for (int j=i+1; j<values.size(); j++) {
5                 if (values.get(j).compareTo(values.get(minIndex)) < 0) {
6                      minIndex = j; }}
7            T temp = values.get(i);
8            values.set(i, values.get(minIndex));
9            values.set(minIndex, temp);
   }
```

در این الگوریتم به کمک دو حلقه تودرتو کوچکترین مقدار از هر خانه‌ی لیست به بعد پیدا می‌شود(خطوط ۱ تا۵) و سپس با عنصر آن خانه جابجا می‌شود(خطوط ۷ تا ۹).

تست برنامه

حال مرتبه زمانی دو الگوریتم پیاده‌شده را با هم در کلاس SortMain مقایسه می‌کنیم. برای اینکار یک لیست یکسان با طول نسبتا زیاد (برای درک بهتر اختلاف زمانی اجرای دو الگوریتم) و با استفاده از تایمر، زمان صرف شده برای هرکدام را نمایش می‌دهیم. انتظار داریم زمان صرف شده در مرتب‌سازی ادغامی (با مرتبه زمانی $O(nlgn)$ و مرتبه حافظه‌ی $O(n)$) کمتر از زمان صرف‌شده در مرتب‌سازی انتخابی (با مرتبه زمانی $O(n^2)$ و مرتبه حافظه‌ی $O(n)$) باشد.

SortsMain

```
    public static void main(String[ ] args) {
        // making two identical lists
1       int size = 100,000;
2       ArrayList<Integer> valuesForMergeSort = new ArrayList<>();
3       ArrayList<Integer> valuesForSelectionSort = new ArrayList<>();
4       Random random = new Random();
5       for (int i=0; i<size; i++) {
6           int value = random.nextInt();
7           valuesForMergeSort.add(value);
8           valuesForSelectionSort.add(value); }
        // Merge Sort test
9       long start = System.currentTimeMillis();
10      Sorts.mergeSort(valuesForMergeSort);
11      System.out.println("mergeSort takes "+(System.currentTimeMillis()-start)+
        "milliseconds to perform.");
        // Selection Sort test
12      start = System.currentTimeMillis();
13      Sorts.selectionSort(valuesForSelectionSort);
14      System.out.println("selectionSort takes "+(System.currentTimeMillis()-start)+
        "milliseconds to perform.");
    }
```

مراجع

[۱] قدسی، محمّد. *داده ساختارها و مبانی الگوریتم‌ها*. تهران: فاطمی، ۱۳۹۵