KSU NOTION

CSC 212

Mid Review

مراجعة للاختبار النصفي

عال ٢١٢

- ADT List
  - Double Linked List
    - ADT Queue
      - Stacks

ADT List Specification

Elements: The elements are of generic type <Type> (The elements are placed in nodes for linked list implementation).

Structure: the elements are linearly arranged. The first element is called head, there is a element called current.

Domain: the number of elements in the list is bounded therefore the domain is finite. Type name of elements in the domain: List

Operations: We assume all operations operate on a list L.

1. **Method** findFirst ( )
   requires: list L is not empty **input:** none
   results: first element set as the current element **output:** none.
2. **Method** findNext ( )
   requires: list L is not empty. Current is not last **input:** none
   results: element following the current element is made the current element
   output: none.
3. **Method** retrieve (Type e)
   requires: list L is not empty **input:** none
   results: current element is copied into e. **output:** element e.
4. **Method** update (Type e).
   requires: list L is not empty **input:** e.
   results: the element e is copied into the current node.
   output: none.
5. **Method** insert (Type e).
   requires: list L is not full **input:** e.
   results: a new node containing element e is created and inserted after the current element in the list. The new element e is made the current element. If the list is empty e is also made the head element.
   output: none.
6. **Method** remove ( )
   requires: list L is not empty **input:** none
   results: the current element is removed from the list. If the resulting list is empty current is set to NULL. If successor of the deleted element exists it is made the new current element otherwise first element is made the new current element
   output: none.
7. **Method** full (boolean flag)
   input: none.
   returns: if the number of elements in L has reached the maximum number allowed then flag is set to true otherwise false.
   output: flag.
8. **Method** empty (boolean flag)

input: none.

results: if the number of elements in **L** is zero then flag is set to true otherwise false.

Output: flag.

9. **Method** last (boolean flag).

input: none **requires:** L is not empty.

Results: if the last element is the current element then flag is set to true otherwise false.

Output: flag

interface (List): Representation

```
public interface List<T> {
	public void findFirst();

	public void findNext();

	public T retrieve();

	public void update(T e);

	public void insert(T e);

	public void remove();

	public boolean full();

	public boolean empty();

	public boolean last();
}
```

class (Linked List): Representation

```
public class LinkedList<T> implements List<T> {
	private Node<T> head;
	private Node<T> current;

	public LinkedList() {
		head = current = null;
	}

	public boolean empty() {
		return head == null;
	}

	public boolean last() {
		return current.next == null;
	}

	public boolean full() {
		return false;
	}

	public void findFirst() {
		current = head;
	}

	public void findNext() {
		current = current.next;
	}
```

```
	public T retrieve() {
		return current.data;
	}

	public void update(T val) {
		current.data = val;
	}

	public void insert(T val) {
		Node<T> tmp;
		if (empty()) {
			current = head = new Node<T>(val);
		} else {
			tmp = current.next;
			current.next = new Node<T>(val);
			current = current.next;
			current.next = tmp;

		}
	}

	public void remove() {
		if (current == head) {
			head = head.next;
		} else {
			Node<T> tmp = head;
			while (tmp.next != current)
				tmp = tmp.next;
			tmp.next = current.next;
		}
		if (current.next == null)
			current = head;
		else
			current = current.next;
	}
	public void findPrevious() {
		Node<T> tmp = head;
		while (tmp.next != current)
			tmp = tmp.next;
		current = tmp;
	}
}
```

class (Array): Representation

```
public class ArrayList<T> implements List<T> {

	private int maxsize;
	private int size;
	private int current;
	private T[] nodes;

	public ArrayList(int n) {
		maxsize = n;
		size = 0;
		current = -1;
		nodes = (T[]) new Object[n];

	}

	public boolean full() {
		return size == maxsize;
	}

	public boolean empty() {
		return size == 0;
```

```
	}

	public boolean last() {
		return current == size - 1;
	}

	public void findFirst() {
		current = 0;
	}

	public void findNext() {
		current++;
	}

	public T retrieve() {
		return nodes[current];
	}

	public void update(T val) {
		nodes[current] = val;
	}

	public void insert(T val) {
		for (int i = size - 1; i > current; --i) {
			nodes[i + 1] = nodes[i];
		}
		current++;
		nodes[current] = val;
		size++;
	}

	public void remove() {
		for (int i = current + 1; i < size; i++) {
			nodes[i - 1] = nodes[i];

			size--;
			if (size == 0)
				current = -1;
			else if (current == size)
				current = 0;
		}
	}
	public void findPrevious() {
			current--;
		}
}
```


Complexity

Operation	Array List	Linked List
Empty	O(1)	O(1)
Last	O(1)	O(1)
Full	O(1)	O(1)
FindFirst	O(1)	O(1)
FindNext	O(1)	O(1)
FindPrevious	O(1)	O(n)
Retrieve	O(1)	O(1)
Update	O(1)	O(1)
Insert	O(n)	O(1)
Remove	O(n)	O(n)

ADT List - Double-Linked List Specification

Elements: The elements are of generic type <Type> (The elements are placed in nodes for linked list implementation).
Structure: the elements are linearly arranged. The first element is called head, there is a element called current.
Domain: the number of elements in the list is bounded therefore the domain is finite. Type name of elements in the domain: List
Operations: We assume all operations operate on a list L.

1. Method FindFirst ( )
   requires: list L is not empty. input: none
   results: first element set as the current element.
   output: none.
2. Method FindNext ( )
   requires: list L is not empty. Cur is not last. input: none
   results: element following the current element is made the current element.
   output: none.
3. Method FindPrevious ( )
   requires: list L is not empty. Cur is not Head. input: none
   results: element Previous to the current element is made the current element.
   output: none.
4. Method Retrieve (Type e)
   requires: list L is not empty. input: none
   results: current element is copied into e. output: element e.
5. Method Update (Type e).
   requires: list L is not empty. input: e.
   results: the element e is copied into the current node.
   output: none.
6. Method Insert (Type e).
   requires: list L is not full. input: e.
   results: a new node containing element e is created and inserted after the current element in the list. The new element e is made the current element. If the list is empty e is also made the head element. output: none.
7. Method Remove ( )
   requires: list L is not empty. input: none
   results: the current element is removed from the list. If the resulting list is empty current is set to NULL. If successor of the deleted element exists it is made the new current element otherwise first element is made the new current element. output: none.
8. Method Full (boolean flag)
   input: none. returns: if the number of elements in L has reached the maximum number

allowed then flag is set to true otherwise false.
output: flag.

9. Method Empty (boolean flag).
   input: none. results: if the number of elements in L is zero, then flag is set to true otherwise false.
   Output: flag.
10. Method First (boolean flag).
    input: none. requires: L is not empty. Results: if the first element is the current element then flag is set to true otherwise false. Output: flag
11. Method Last (boolean flag).
    input: none. requires: L is not empty. Results: if the last element is the current element then flag is set to true otherwise false. Output: flag
12. Method FindLast ( )
    requires: list L is not empty. input: none
    results: last element is set as the current element. output: none.

class (Node): Representation

```
public class Node<T> {

	public T data;
	public Node<T> next;

	public Node() {
		data = null;
		next = null;
	}

	public Node(T val) {
		data = val;
		next = null;
	}
}
```

class (Double-Linked List): Representation

```
public class DoubleLinkedList<T> {
	private Node<T> head;
	private Node<T> current;

	public DoubleLinkedList() {
		head = current = null;
	}

	public boolean empty() {
		return head == null;
	}

	public boolean last() {
		return current.next == null;
	}

	public boolean first() {
		return current.previous == null;
	}

	public boolean full() {
		return false;
	}
```

```
	public void findFirst() {
		current = head;
	}

	public void findNext() {
		current = current.next;
	}

	public void findPrevious() {
		current = current.previous;
	}

	public T retrieve() {
		return current.data;
	}

	public void update(T val) {
		current.data = val;
	}

	public void insert(T val) {
		Node<T> tmp = new Node<T>(val);
		if (empty()) {
			current = head = tmp;
		} else {
			tmp.next = current.next;
			tmp.previous = current;
			if (current.next != null)
				current.next.previous = tmp;
			current.next = tmp;
			current = tmp;
		}
	}

	public void remove() {
		if (current == head) {
			head = head.next;
			if (head != null)
				head.previous = null;
		} else {
			current.previous.next = current.next;
			if (current.next != null)
				current.next.previous = current.previous;
		}

		if (current.next == null)
			current = head;
		else
			current = current.next;
	}

// Another simpler implementation for remove (optional)
	public void remove2() {
		// if current is first only move right (no node before it)
		// otherwise (there is a node before it) connect previous with next
		if (current == head)
			head = head.next;
		else
			current.previous.next = current.next;

		// if current is not last (there is a node after it), then connect next with
		// previous
		if (current.next != null)
			current.next.previous = current.previous;
```

```
                // move current either to first (when it is last)
                // otherwise, move it next
                if (current.next == null)
                        current = head;
                else
                        current = current.next;
        }
        public void findLast() {
                while(current.next != null)
                         current = current.next;
        }

}
```

Complexity

Operation	Array List	Linked List	Double-Linked List
Empty	O(1)	O(1)	O(1)
Last	O(1)	O(1)	O(1)
Full	O(1)	O(1)	O(1)
FindFirst	O(1)	O(1)	O(1)
FindNext	O(1)	O(1)	O(1)
FindPrevious	O(1)	O(n)	O(1)
Retrieve	O(1)	O(1)	O(1)
Update	O(1)	O(1)	O(1)
Insert	O(n)	O(1)	O(1)
Remove	O(n)	O(n)	O(1)

ADT Queue Specification

Elements: The elements are of generic type <Type> (The elements are placed in nodes for linked list implementations).

Structure: the elements are linearly arranged, and ordered according to the order of arrival. Most recently arrived element is called the back or tail, and least recently arrived element is called the front or head.

Domain: the number of elements in the queue is bounded therefore the domain is finite. Type of elements: Queue

Operations:

1. **Method** Enqueue (Type e)
   requires: Queue Q is not full. **input:** Type e.
   results: Element e is added to the queue at its tail. **output:** none.
2. **Method** Serve (Type e)
   requires: Queue Q is not empty. **input**: none
   results: the element at the head of Q is removed and its value assigned to e. **output**: Type e.
3. **Method** Length (int length)
   requires: none. **input**: none
   results: The number of element in the Queue Q is returned. **output**: length.
4. **Method** Full (boolean flag).

requires: none. **input**: none

results: If Q is full then flag is set to true, otherwise flag is set to false.

output: flag.

5. Method enquiry(Type e)

requires: none. **input**: none

results: returns the data at the head of the queue without changing it.

Output: Type e.

Interface (Queue): Representation

```
public interface Queue<T> {
	public T serve();

	public void enqueue(T e);

	public int length();

	public boolean full();
}
```

class (Node): Representation

```
public class Node<T> {
	public T data;
	public Node<T> next;

	public Node() {
		data = null;
		next = null;
	}

	public Node(T val) {
		data = val;
		next = null;
	}
}
```

class (LinkedQueue): Representation

```
public class LinkedQueue<T> implements Queue<T> {
	private Node<T> head, tail;
	private int size;

	/** Creates a new instance of LinkedQueue */
	public LinkedQueue() {
		head = tail = null;
		size = 0;
	}

	public boolean full() {
		return false;
	}

	public int length() {
		return size;
	}

	public void enqueue(T e) {
		if (tail == null) {
			head = tail = new Node<T>(e);
		} else {
```

```
                tail.next = new Node<T>(e);
                tail = tail.next;
            }
            size++;
        }

        public T serve() {
            T x = head.data;
            head = head.next;
            size--;
            if (size == 0)
                tail = null;
            return x;
        }

        public T enquiry() {
            return head.data;
        }

}
```

class (ArrayQueue): Representation

```
public class ArrayQueue<T> implements Queue<T> {
        private int maxsize;
        private int size;
        private int head, tail;
        private T[] nodes;

        public ArrayQueue(int n) {
            maxsize = n;
            size = 0;
            head = tail = 0;
            nodes = (T[]) new Object[n];
        }

        public boolean full() {
            return size == maxsize;
        }

        public int length() {
            return size;
        }

        public void enqueue(T e) {
            nodes[tail] = e;
            tail = (tail + 1) % maxsize;
            size++;
        }

        public T serve() {
            T e = nodes[head];
            head = (head + 1) % maxsize;
            size--;
            return e;
        }

        public static <T> T enquiry(Queue<T> q) {
            T data = q.serve();
            q.enqueue(data);
            for (int i = 0; i < q.length() - 1; i++)
                q.enqueue(q.serve());
            return data;
        }

        public T enquiry() {
```

```
        return nodes[head];
}}
```

Method enquiry(Type e) As a User

requires: none. **input**: none

results: returns the data at the head of the queue without changing it.

Output: Type e.

```
public static<T> T enquiry(Queue<T> q) {
        T data = q.serve();
        q.enqueue(data);
        for(int i = 0; i < q.length() - 1; i++)
                q.enqueue(q.serve());
                return data;

}
```

ADT Priority Queue Specification

- Each data element has a priority associated with it. Highest priority item is served first.
- Real World Priority Queues: hospital emergency rooms (most sick patients treated first), events in a computer system, etc.
- Priority Queue can be viewed as:
    - View 1: Priority queue as an ordered list.
    - View 2: Priority queue as a set.
- **Elements**: The elements are of type PQNode. Each node has in it a data element of generic type <Type> and a priority of type Priority (which could be int type).
- **Structure**: the elements are linearly arranged, and may be ordered according to a priority value, highest priority element is called the front or head.
- **Domain**: the number of nodes in the queue is bounded therefore the domain is finite. Type of elements: PriorityQueue
- **Operations:**

1. **Method** Enqueue (Type e, Priority p)
   requires: PQ is not full. **input:** e, p.
   results: Element e is added to the queue according to its priority. **output:** none.
2. **Method** Serve (PQElement<Type> pqe)
   requires: PQ is not empty. **input**: None
   results: the element and the priority at the head of PQ is removed and returned.
   output: pqe.
3. **Method** Length (int length)
   input: **results**: The number of element in the PQ is returned. **output**: length.
4. **Method** Full (boolean flag).
   requires: **input**:
   results: If PQ is full then flag is set to true, otherwise flag is set to false. **output**: flag.

class (PQNode): Representation

```
public class PQNode<T> {
        public T data;
        public int priority;
```

```
        public PQNode<T> next;

        public PQNode() {
                next = null;
        }

        public PQNode(T e, int p) {
                data = e;
                priority = p;
        }
}
```

class (LinkedPQ): Representation

```
public class LinkedPQ<T> {
        private int size;
        private PQNode<T> head;

        public LinkedPQ() {
                head = null;
                size = 0;
        }

        public int length() {
                return size;
        }

        public boolean full() {
                return false;
        }

        public void enqueue(T e, int pty) {
                PQNode<T> tmp = new PQNode<T>(e, pty);
                if ((size == 0) || (pty > head.priority)) {
                        tmp.next = head;
                        head = tmp;
                } else {
                        PQNode<T> p = head;
                        PQNode<T> q = null;
                        while ((p != null) && (pty <= p.priority)) {
                                q = p;
                                p = p.next;
                        }
                        tmp.next = p;
                        q.next = tmp;
                }
                size++;
        }

        public PQElement<T> serve() {
                PQNode<T> node = head;
                PQElement<T> pqe = new PQElement<T>(node.data, node.priority);
                head = head.next;
                size--;
                return pqe;
        }

}

class PQElement<T> {
        public T data;
        public int p;

        public PQElement(T e, int pr) {
                data = e;
                p = pr;
```

```
        }
}
```

Complexity

Operation	Queue (LL)	Queue (CA)	Priority Queue (LL)	Priority Queue (CA)
Full	O(1)	O(1)	O(1)	O(1)
Length	O(1)	O(1)	O(1)	O(1)
Enqueue	O(1)	O(1)	O(n)	O(n)
Serve	O(1)	O(1)	O(1)	O(1)

ADT Double-Ended Queues Specification

- Double ended queue (or a deque) supports insertion and deletion at both the front and the tail of the queue.
- The first element is called head and the last element is called tail.
- Supports operations: addFirst( ), addLast(), removeFirst( ) and removeLast( ).
- Can be used in place of a queue or a stack.
- **Operations:** (Assume all operations are performed on deque DQ)

1. Method addFirst (Type e)
   requires: DQ is not full. input: e.
   results: Element e is added to DQ as first element. output: none.
2. Method addLast (Type e)
   requires: DQ is not full. input: e
   results: Element e is added to DQ as last element. output: none.
3. Method removeFirst (Type e)
   requires: DQ is not empty. input: none results: Removes and returns the first element of DQ. output: e.
4. Method removeLast (Type e)
   requires: DQ is not empty. input: none.
   results: Removes and returns the last element of DQ. output: e.
5. Method getFirst (Type e)
   requires: DQ is not empty. input: none
   results: Returns the first element of DQ. output: e.
6. Method getLast (Type e)
   requires: DQ is not empty. input: none
   results: Returns the last element of DQ. output: e
7. Method size (int x)
   input: none results: Returns the number of elements in DQ. output: x
8. Method empty (boolean x)
   input: none results: if DQ is empty returns x as true otherwise false. output: x

Complexity

Operation	Double-Ended Queue (LL)	Double-Ended Queue (CA)	Double-Ended Queue (DLL)
AddFirst	O(n)	O(1)	O(1)
AddLast	O(1)	O(1)	O(1)
RemoveFirst	O(1)	O(1)	O(1)
RemoveLast	O(n)	O(1)	O(1)
GetFirst	O(1)	O(1)	O(1)
GetLast	O(1)	O(1)	O(1)
Size	O(1)	O(1)	O(1)
Empty	O(1)	O(1)	O(1)

ADT Stack Specification

Elements: The elements are of a generic type <Type>. (In a linked implementation an element is placed in a node)

Structure: the elements are linearly arranged, and ordered according to the order of arrival, most recently arrived element is called top.

Domain: the number of elements in the stack is bounded therefore the domain is finite. Type of elements: Stack

Operations: All operations operate on a stack S.

1. Method push (Type e)
   requires: Stack S is not full.
   input: Type e.
   results: Element e is added to the stack as its most recently added elements. output: none.
2. Method pop (Type e)
   requires: Stack S is not empty.
   input: none
   results: the most recently arrived element in S is removed and its value assigned to e.
   output: Type e.
3. Method empty (boolean flag)
   input: none
   results: If Stack S is empty then flag is true, otherwise false.
   output: flag.
4. Method Full (boolean flag).
   requires: none
   input: none
   results: If S is full then Full is true, otherwise Full is false.
   output: flag.

Interface (Stack): Representation

```
public interface Stack<T> {
	public T pop();

	public void push(T e);

	public boolean empty();

	public boolean full();
}
```

class (Node): Representation

```
public class Node<T> {
	public T data;
	public Node<T> next;

	public Node() {
		data = null;
		next = null;
	}

	public Node(T val) {
		data = val;
		next = null;
	}
}
```

class (LinkedStack): Representation

```
public class LinkedStack<T> implements Stack<T> {
	private Node<T> top;

	public LinkedStack() {
		top = null;
	}

	public boolean empty() {
		return top == null;
	}

	public boolean full() {
		return false;
	}

	public void push(T e) {
		Node<T> tmp = new Node<T>(e);
		tmp.next = top;
		top = tmp;
	}

	public T pop() {
		T e = top.data;
		top = top.next;
		return e;
	}
}
```

class (ArrayStack): Representation

```
public class ArrayStack<T> implements Stack<T> {
	private int maxsize;
	private int top;
	private T[] nodes;
```

```
        public ArrayStack(int n) {
                maxsize = n;
                top = -1;
                nodes = (T[]) new Object[n];
        }

        public boolean empty() {
                return top == -1;
        }

        public boolean full() {
                return top == maxsize - 1;
        }

        public void push(T e) {
                nodes[++top] = e;
        }

        public T pop() {
                return nodes[top--];
        }
}
```