자료구조

L05 Doubly Linked Lists, Stacks, and Queues

2022년 1학기

국민대학교 소프트웨어학부

Summary

- **리스트 순회, Iterator**
- 이중연결리스트(Doubly Linked List)
- 스택(Stack), 큐(Queue)
- 사전(Dictionary)

리스트 순회

- List에 있는 값들을 **순회**하려면?

```
List<Integer> myList = new ArrayList<>(); (혹은 new LinkedList<>();)

     ...

for(int i = 0; i < myList.length(); i++)
    System.out.println(myList.getValue(i));
```

- ArrayList일 때 시간복잡도?

- LinkedList일 때 시간복잡도?

리스트 순회

- LinkedList를 좀 더 효율적으로 순회 하려면?

```
Link<Integer> curr = ((LinkedList<Integer>) myList).head();
Link<Integer> tail = ((LinkedList<Integer>) myList).tail();

while(curr != tail) {
    curr = curr.next();
    System.out.println(curr.item());
}
```

- 이 방식의 **문제점**
  - 형변환(type cast)을 해야되네? 코드도 복잡하네?
    - ArrayList, LinkedList 상관 없이, List로서 사용하고 싶은데...?
    - 이럴거면 ADT나 Interface를 왜 씀?

리스트 순회, Iterator

- **Iterator 패턴**을 이용하자!
  - 이제 ArrayList든 LinkedList든 상관없이 효율적으로 순회 가능!

```
List<Integer> myList = new LinkedList<>();
    ...
ListIterator<Integer> itr = myList.listIterator();
while(itr.hasNext()) {
    int item = itr.next();
    System.out.println(item);
}
```

- ListIterator ADT

```
public interface ListIterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
}
```

리스트 순회, Iterator

- **listIterator()** 메소드가 추가된 **List** interface

```
public interface List<E> {
    public void clear();
    public void insert(int pos, E item);
    public void append(E item);
    public void update(int pos, E item);
    public E getValue(int pos);
    public E remove(int pos);
    public int length();
    public ListIterator<E> listIterator();
}
```

리스트 순회, Iterator

- ArrayList의 `listIterator()`

```
public ListIterator<E> listIterator() {
    return new ListIterator<E>() {
        int pos = 0;

        public boolean hasNext() { return pos < size; }

        public E next() { return data[pos++]; }

        public boolean hasPrevious() { return pos > 0; }

        public E previous() { return data[--pos]; }
    };
}
```

리스트 순회, Iterator

- LinkedList의 `listIterator()`

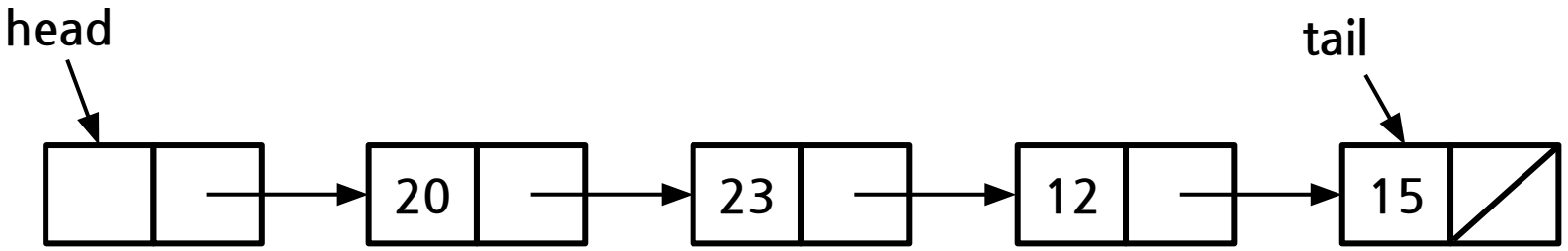
```
public ListIterator<E> listIterator() {
    return new ListIterator<E>() {
        Link<E> curr = head;
        public boolean hasNext() { return curr != tail; }
        public E next() {
            curr = curr.next();
            return curr.item();
        }
        public boolean hasPrevious() { return curr != head; }
        public E previous() {
            Link<E> prev = head;
            while(prev.next() != curr)
                prev = prev.next();
            curr = prev;
            return curr.next().item();
        }
    };
}
```

Summary

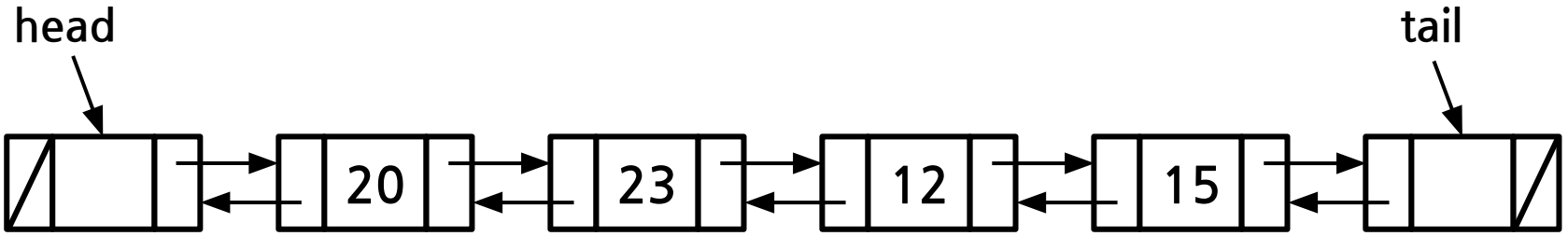
- 리스트 순회, Iterator
- **이중연결리스트(Doubly Linked List)**
- 스택(Stack), 큐(Queue)
- 사전(Dictionary)

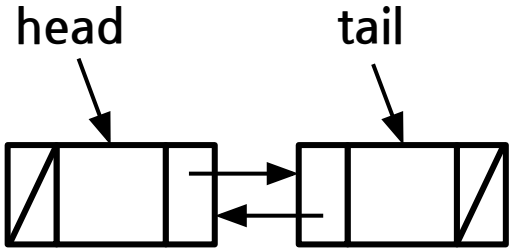
연결리스트Linked Lists의 한계

- ListIterator에서 `previous()`의 시간복잡도는..?
- 반대 방향 순회의 시간 복잡도는?
- 어떻게 반대 방향 순회를 빠르게 할 수 있을까?
  - Idea: **이중 연결 리스트**Doubly Linked Lists!

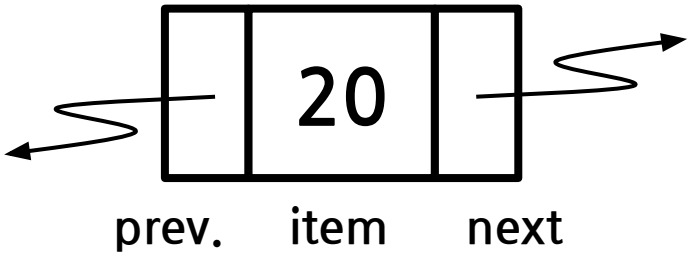


이중연결리스트 Doubly Linked Lists





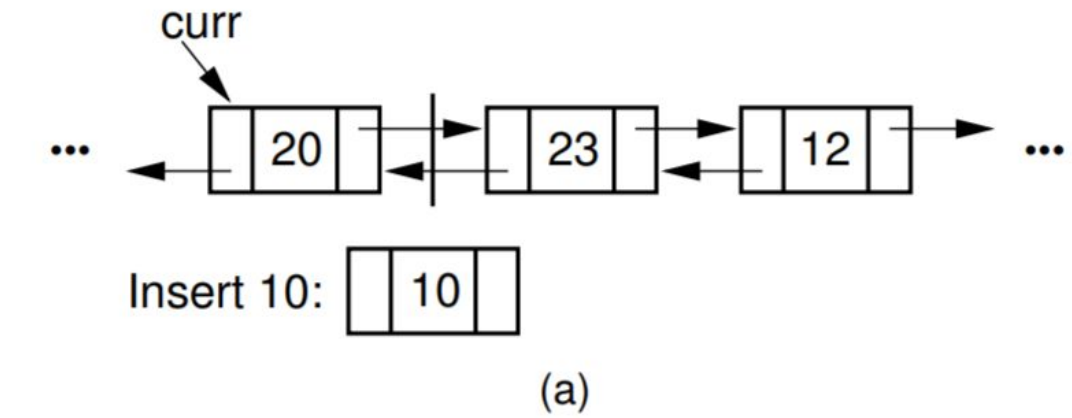
이중연결리스트 Doubly Linked Lists



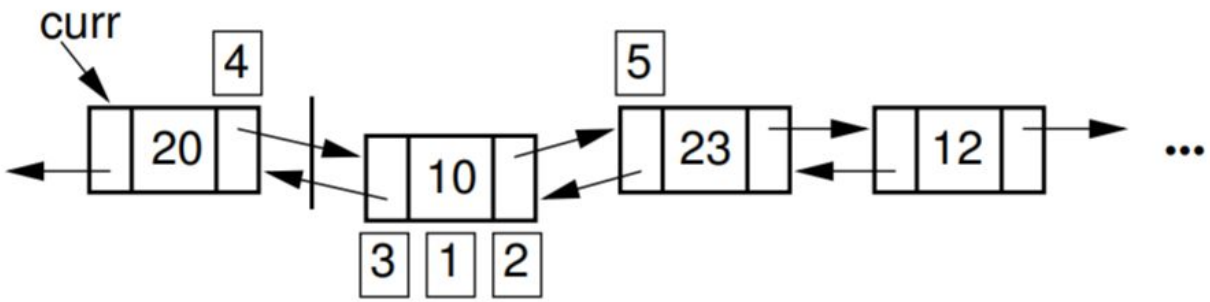

```
class DLink<E> {
    private E item;
    private DLink<E> next;
    private DLink<E> prev;
    DLink(E item, DLink<E> prev, DLink<E> next) {
        this.item = item; this.prev = prev; this.next = next;
    }
    DLink<E> next() { return next; }
    DLink<E> setNext(DLink<E> next) { return this.next = next; }
    DLink<E> prev() { return prev; }
    DLink<E> setPrev(DLink<E> prev) { return this.prev = prev; }
    E item() { return item; }
    E setItem(E item) { return this.item = item; }
}
```

Doubly Linked Lists: Insert

- 20과 23사이에 10 넣기



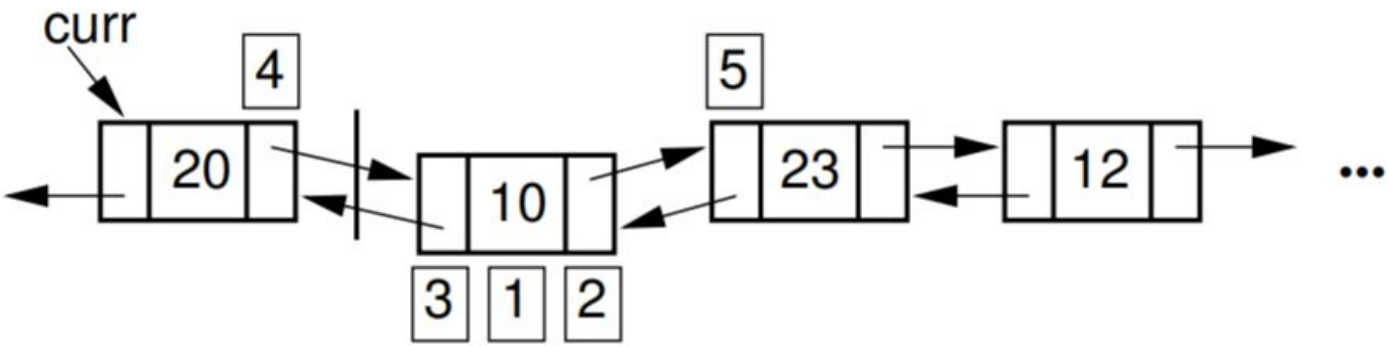
(a)



(b)

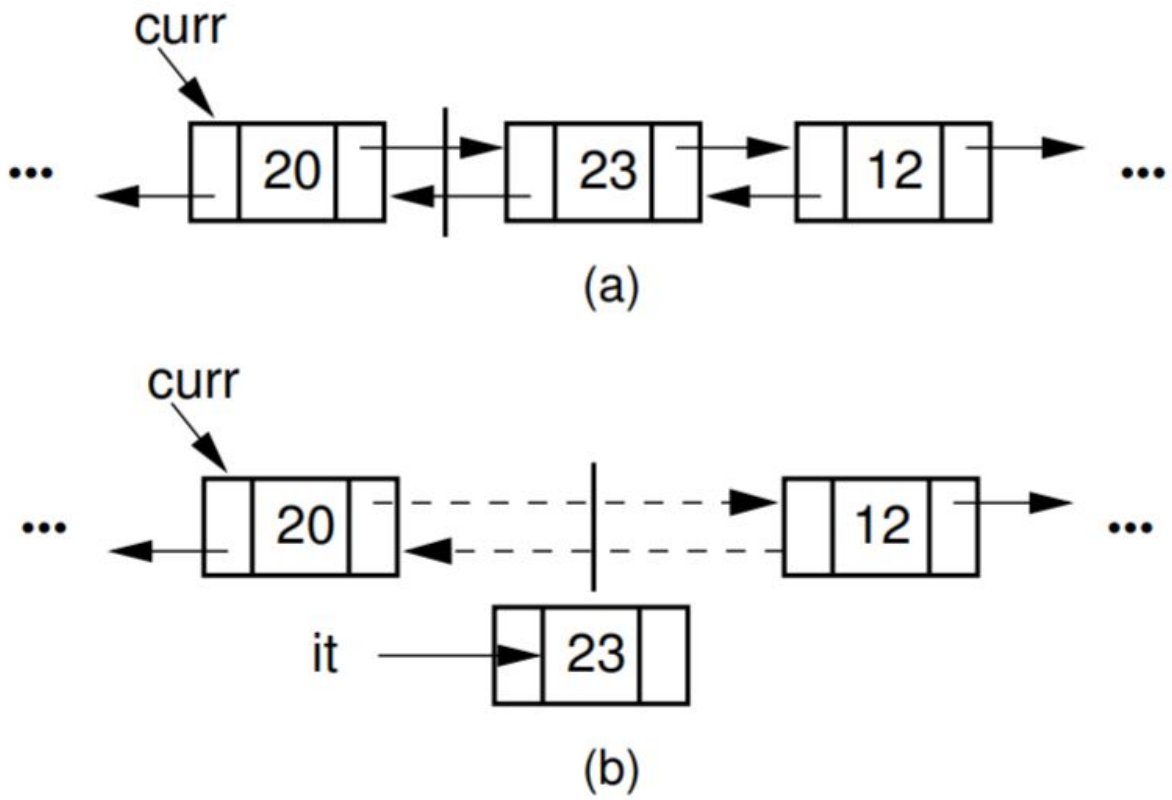
Doubly Linked Lists: Insert

```
public void insert(int pos, E item) {
    Link<E> curr = head;
    for(int i=0; i<pos; i++)
        curr = curr.next();

        4          1          3          2
    curr.setNext(new DLink<E>(item, curr, curr.next()));
    curr.next().next().setPrev(curr.next());
    size++;                    5
}
```



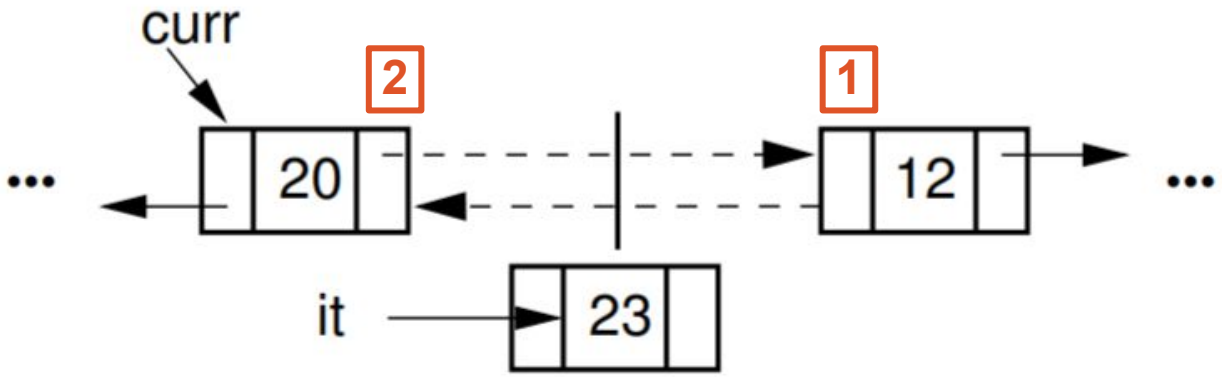
Doubly Linked Lists: Remove

- 23 지우기



Doubly Linked Lists: Remove

```
public E remove(int pos) {
    Link<E> curr = head;
    for(int i=0; i<pos; i++)
        curr = curr.next();

    E ret = curr.next().item();
    curr.next().next().setPrev(curr);
    curr.setNext(curr.next().next());
    cnt--;
    return ret;
}
```



별도의 Head와 Tail

- 이중연결리스트는 별도의 tail을 갖음
  - 이유: 예외처리할 필요 없음

- `insert()` with fixed Tail node

```
curr.setNext(new DLink<E>(item, curr, curr.next()));
curr.next().next().setPrev(curr.next());
```

- `insert()` without fixed Tail node

```
curr.setNext(new DLink<E>(item, curr, curr.next()));
curr.next().next().setPrev(curr.next());
if(tail == curr) tail = curr.next();
```

별도의 Head와 Tail

- remove() with fixed Tail node

```
E ret = curr.next().item();
curr.next().next().setPrev(curr);
curr.setNext(curr.next().next());
```

- remove() without fixed Tail node

```
E ret = curr.next().item();
if (tail == curr.next()) tail = curr;
else curr.next().next().setPrev(curr);
curr.setNext(curr.next().next());
```

Summary

- 리스트 순회, Iterator
- 이중연결리스트(Doubly Linked List)
- **스택(Stack), 큐(Queue)**
- 사전(Dictionary)

Stacks

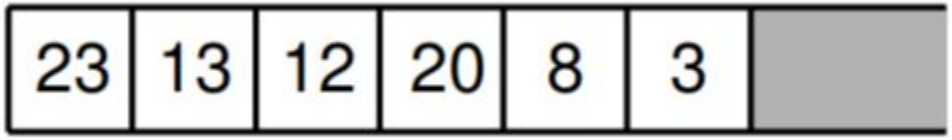
- LIFO: Last In, First Out
- **리스트의 제한된 형태**: 넣고 빼기가 리스트의 한쪽 끝에서만 가능
- Notation:
  - PUSH - 넣기
  - POP - 빼기
  - TOP - 맨 위의 값 리턴

Stack ADT

```
public interface Stack<E> {
    /** Reinitialize the stack. */
    public void clear();

    /** Push an element onto the top of the stack.
    @param it The element being pushed onto the stack. */
    public void push(E it);

    /** Remove and return the element at the top of the stack.
    @return The element at the top of the stack. */
    public E pop();

    /** @return A copy of the top element. */
    public E topValue();

    /** @return The number of elements in the stack. */
    public int length();
}
```

Array-based Stack

```
public class AStack<E> implements Stack<E>{
    private int maxSize; // Max size of stack
    private int top; // Index for top
    private E [] listArray;
    ...
}
```



- **이슈:**
  - 어느쪽이 탑일까?
  - 각 연산의 비용은?
    - PUSH, POP, TOP

Linked Stack

```
class LStack<E> implements Stack<E> {
    private Link<E> top;
    private int size;
    ...
}
```

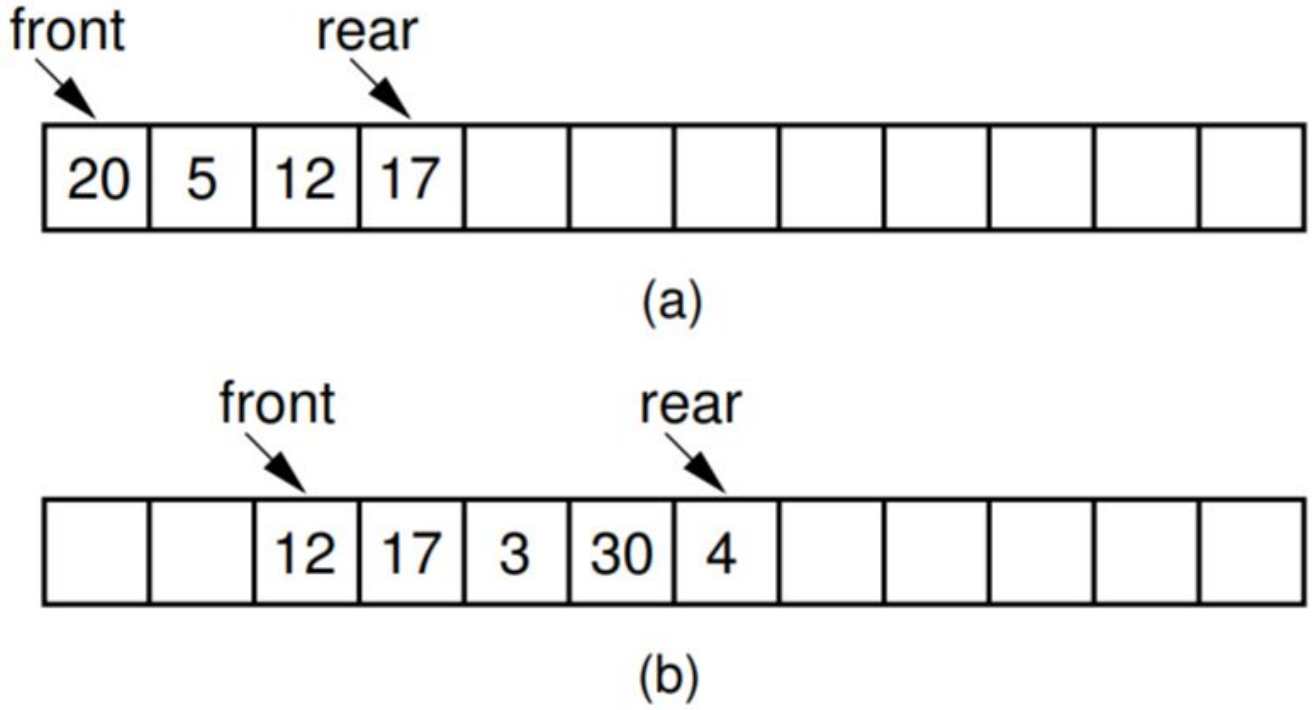
- 각 연산의 비용은?
  - clear, push, pop, topValue, length

- 배열 기반 Stack 구현과 비교하여, 링크 기반 Stack의 메모리 사용량은?

Queues

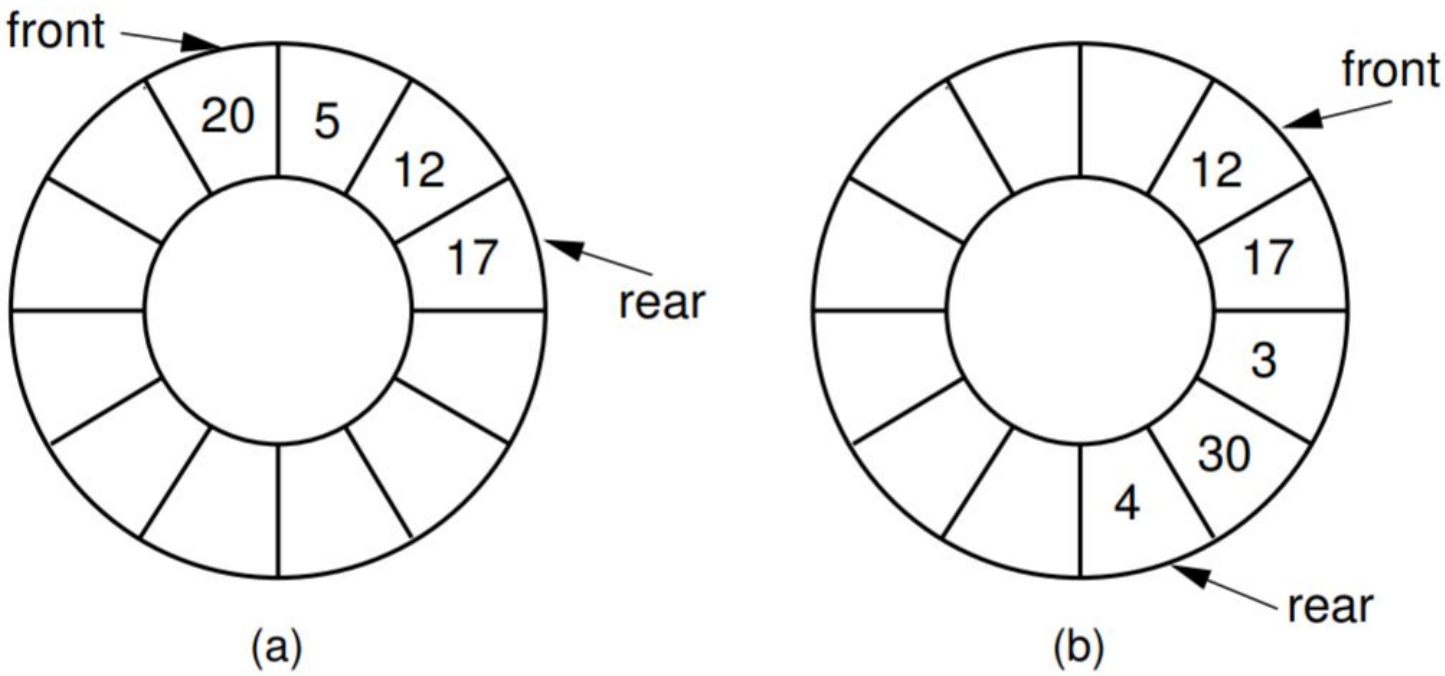
- LIFO: Last In, First Out
- **리스트의 제한된 형태**:
  한쪽 끝에서만 넣고, 다른쪽 끝에서만 뺌
- Notation:
  - **Enqueue** - 넣기
  - **Dequeue** - 빼기
  - **Front** - 맨 앞의 값
  - **Rear** - 맨 뒤의 값

Array-based Queue



- 계속해서 사용하면, 배열의 한쪽 끝으로 점점 이동하는데…
  - 어떻게 하면 영구적으로 쓸 수 있을까?

Array-based Queue



"Circular" Queue

Summary

- 리스트 순회, Iterator
- 이중연결리스트(Doubly Linked List)
- 스택(Stack), 큐(Queue)
- **사전(Dictionary)**

Dictionary

- 값을 넣고, 지우고, 검색을 하려는데… 값이 만일 숫자나 문자가 아니라 좀 복잡하다면? 무엇으로 검색하지…?
  - 예) 각 아이템이 **레코드**Records (학번, 이름, 나이, 성적) 인 경우
- 필요한 개념
  - 키(Key)로 검색: 검색 기준을 정의
    예) (학번, 이름, 나이, 성적)의 경우 '학번' 으로 검색
- 키 비교: 같음, 상대적 순서

레코드Records와 키Keys

- **문제**: 레코드에서 어떻게 키를 뽑을까?
  - 레코드는 여러 개의 키를 가질 수 있음
  - 키는 레코드 내에서만 쓰이지 않고, 검색 등에도 쓰임
- **방법**: **키를 따로 빼서** 레코드와 **쌍**Pair으로 저장
  - 즉, (키, 레코드) 쌍 혹은 (키, 값) 쌍 이라고도 함

Dictionary ADT

```
public interface Dictionary<Key, E> {
    /** Reinitialize dictionary */
    public void clear();

    /** Insert a record */
    public void insert(Key k, E e);

    /** Remove and return a record (null if none exists). */
    public E remove(Key k);

    /** Remove and return an arbitrary record from dictionary. */
    public E removeAny();

    /** Return a record matching "k" (null if none exists). */
    public E find(Key k);

    /** @return The number of records in the dictionary. */
    public int size();
};
```

Payroll Class

```
//Simple payroll entry: ID, name, address
class Payroll {
    private Integer ID;
    private String name;
    private String address;
    Payroll(int inID, String inname, String inaddr)
    {
        ID = inID;
        name = inname;
        address = inaddr;
    }
    public Integer getID() { return ID; }
    public String getname() { return name; }
    public String getaddr() { return address; }
}
```

Dictionary 사용하기

```
//IDdict organizes Payroll records by ID
Dictionary<Integer, Payroll> IDdict =
                        new UALdictionary<Integer, Payroll>();

//namedict organizes Payroll records by name
Dictionary<String, Payroll> namedict =
                        new UALdictionary<String, Payroll>();

Payroll foo1 = new Payroll(5, "Joe", "Anytown");
Payroll foo2 = new Payroll(10, "John", "Mytown");

IDdict.insert(foo1.getID(), foo1);
IDdict.insert(foo2.getID(), foo2);
namedict.insert(foo1.getname(), foo1);
namedict.insert(foo2.getname(), foo2);

Payroll findfoo1 = IDdict.find(5);
Payroll findfoo2 = namedict.find("John");
```

Sorted vs. Unsorted Array List Dictionaries

- Dictionary를 정렬된 Array List를 사용한다면…
  - **이진탐색(Binary Search)**를 통해 **탐색 속도를 올릴 수 있다**
  - **정렬된 상태를 유지**해야 하기 때문에 **값을 삽입할 때 느리다**

- 무엇이 더 나을까?
  - 탐색할 일이 많다면 정렬된 Array List를 쓰는게 Good!
  - 값을 삽입/삭제할 일이 많다면 정렬하는게 딱히…

Questions?