Data Structure

Lecture#8,9: Lists, Stacks, and Queues (Chapter 4)

U Kang
Seoul National University

In This Lecture

- Learn the motivation and main idea of doubly linked list
- Learn the Stack and Queue data structure
- Learn the Dictionary data structure

Limitation of Linked List

- What is the time complexity of prev() in Linked List?
- How can we make prev() fast?

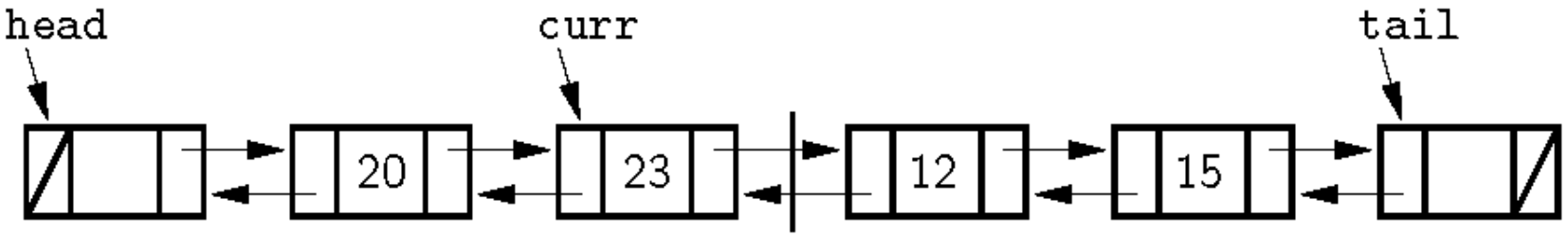
Limitation of Linked List

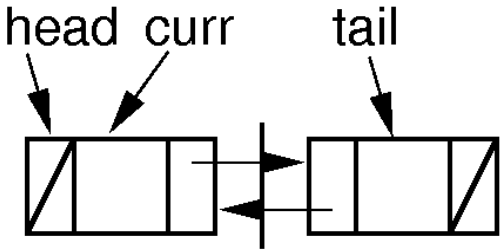
- What is the time complexity of prev() in Linked List?
- How can we make prev() fast?
  - Idea: doubly linked list!

Doubly Linked Lists

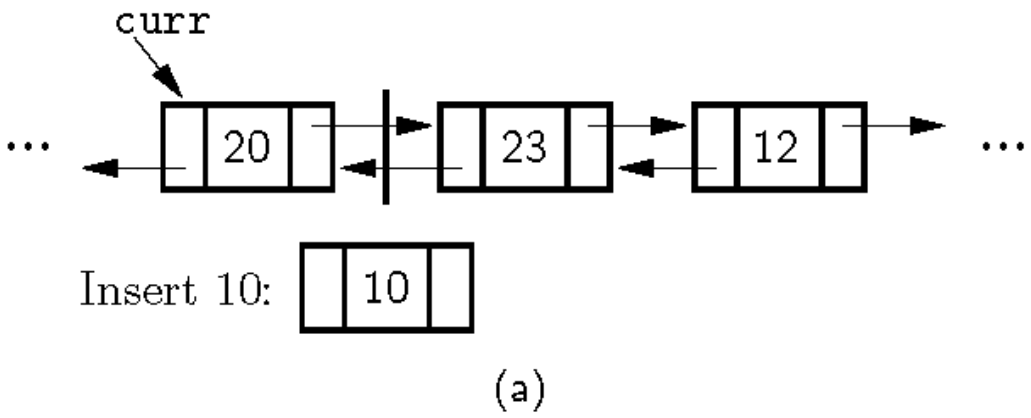
```
class DLink<E> {
  private E element;
  private DLink<E> next;
  private DLink<E> prev;

  DLink(E it, DLink<E> p, DLink<E> n)
    { element = it; prev = p; next = n; }
  DLink(DLink<E> p, DLink<E> n)
    { prev = p; next = n; }

  DLink<E> next() { return next; }
  DLink<E> setNext(DLink<E> nextval)
    { return next = nextval; }
  DLink<E> prev() { return prev; }
  DLink<E> setPrev(DLink<E> prevval)
    { return prev = prevval; }
  E element() { return element; }
  E setElement(E it) { return element = it; }
}
```

Doubly Linked Lists

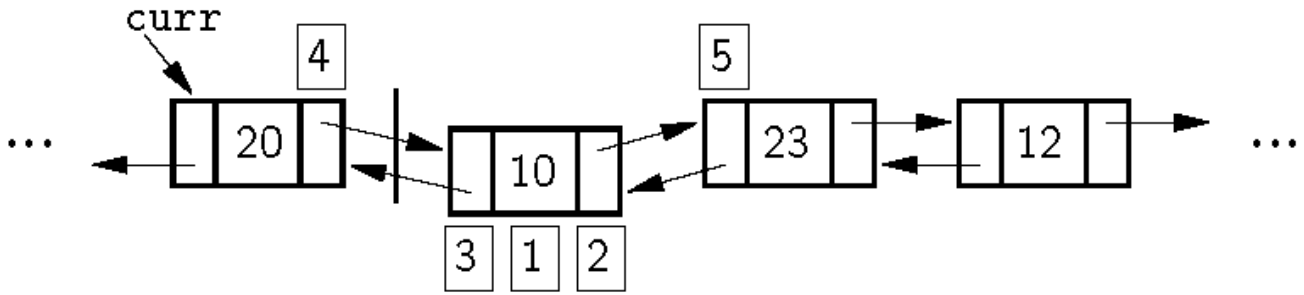




Doubly Linked Insert



(a)

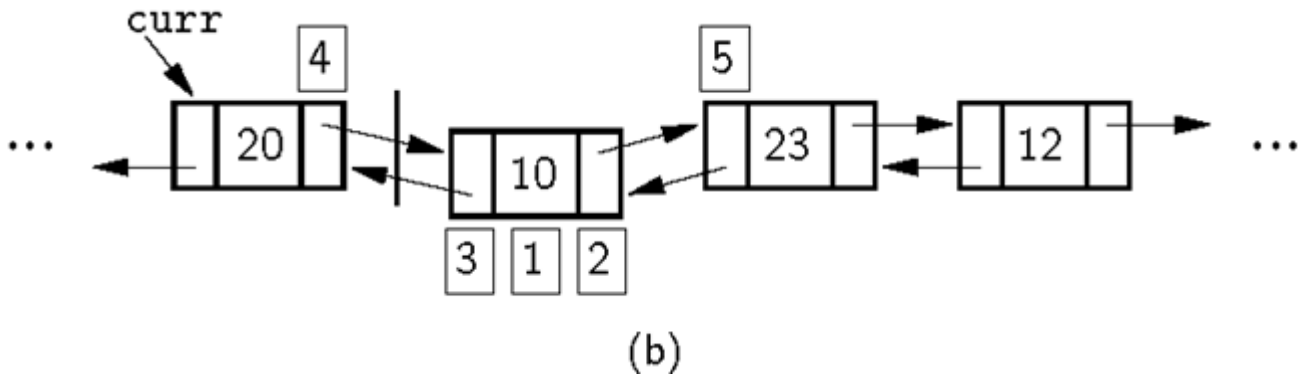


(b)

Doubly Linked Insert

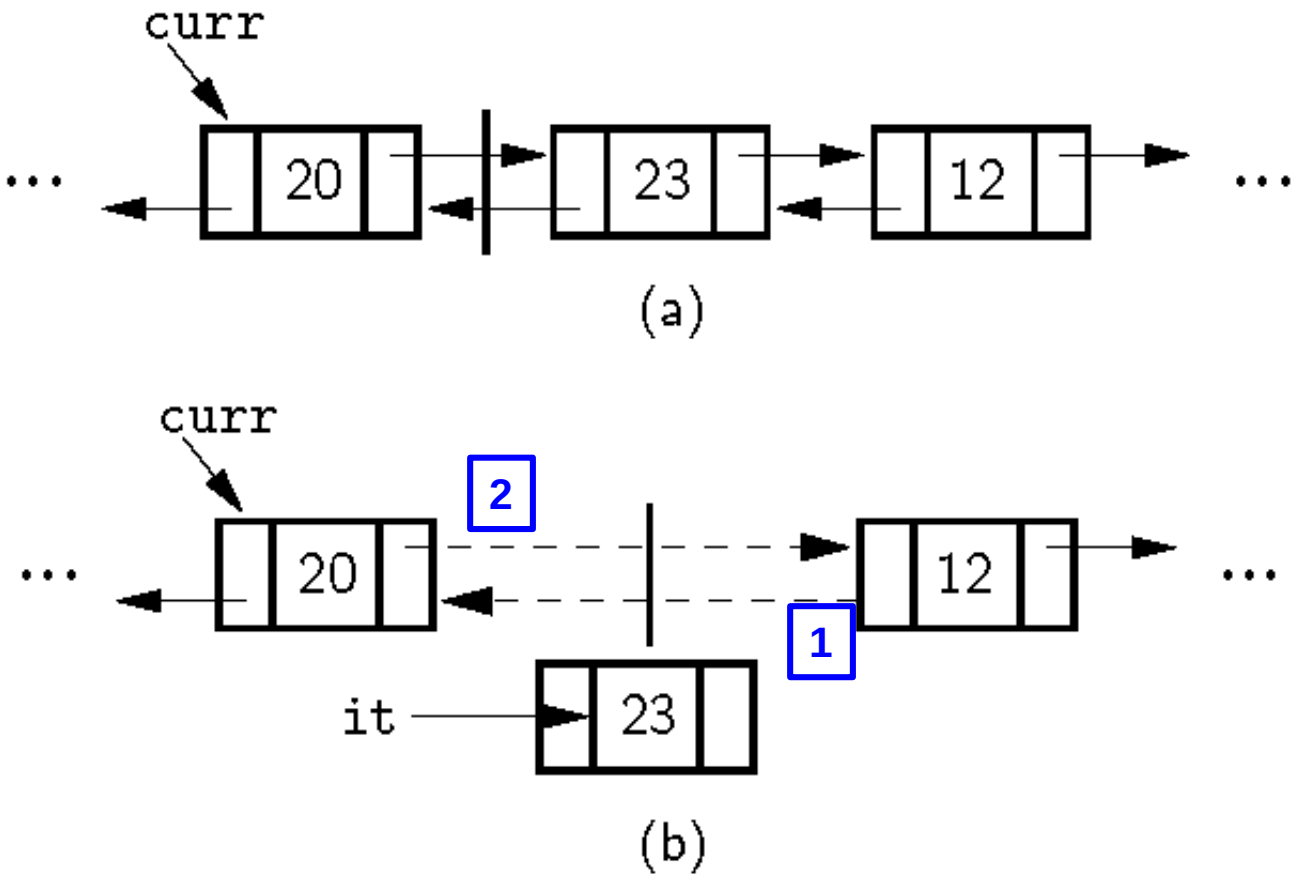
```
public void insert(E it) {
  curr.setNext(
      new DLink<E>(it, curr, curr.next()));
  curr.next().next().setPrev(curr.next());
  cnt++;
}
```

4 3 2 1 5



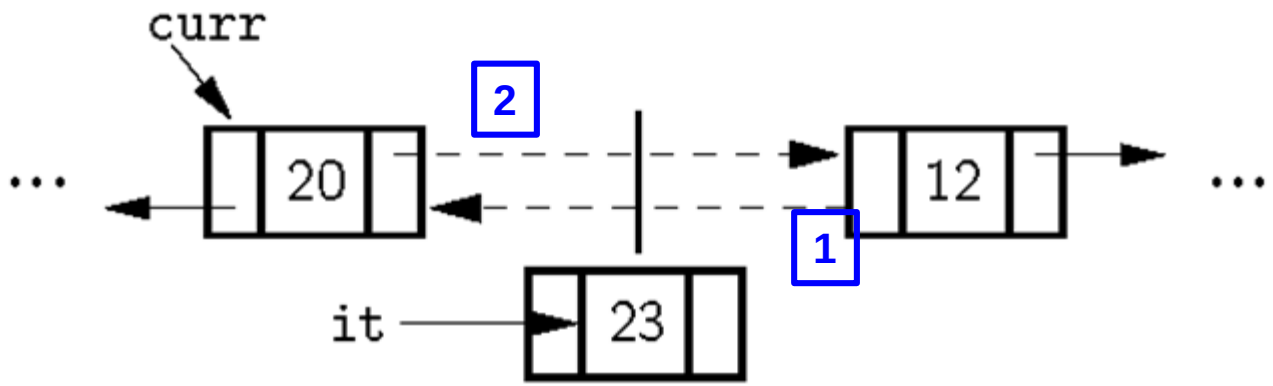
(b)

Doubly Linked Remove



Doubly Linked Remove

```
public E remove() {
  if (curr.next() == tail) return null;
  E it = curr.next().element();
  curr.next().next().setPrev(curr);
  curr.setNext(curr.next().next());
  cnt--;
  return it;
}
```



(b)

Effect of Fixed Head & Tail

- Doubly Linked List has a fixed tail node
  - Reason: no need to consider special cases

- Insert() **with** fixed Tail node

```
public void insert(E it) {
  curr.setNext(new DLink<E>(it, curr, curr.next()));
  curr.next().next().setPrev(curr.next());
  cnt++;
}
```

- Insert() **without** fixed Tail node

```
public void insert(E it) {
  curr.setNext(new DLink<E>(it, curr, curr.next()));
  curr.next().next().setPrev(curr.next());
  if (tail == curr) tail = curr.next();
  cnt++;
}
```

Effect of Fixed Head & Tail

- Remove() **with** fixed Tail node

```
public E remove() {
  if (curr.next() == tail) return null;
  E it = curr.next().element();
  curr.next().next().setPrev(curr);
  curr.setNext(curr.next().next());
  cnt--;
  return it;
}
```

- Remove() **without** fixed Tail node

```
public E remove() {
  if (curr.next() == null) return null;
  E it = curr.next().element();
  if (tail == curr.next()) tail = curr;
  curr.next().next().setPrev(curr);
  curr.setNext(curr.next().next());
  cnt--;
  return it;
}
```

Stacks

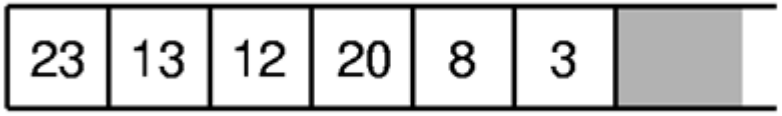
- LIFO: Last In, First Out.
- Restricted form of list: Insert and remove only at front of list.
- Notation:
  - **PUSH** for insertion
  - **POP** for removal
  - **TOP** for the accessible element

Stack ADT

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

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

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

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

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

Array-Based Stack

```
// Array-based stack implementation
private int maxSize;  // Max size of stack
private int top;      // Index for top
private E [] listArray;
```



- Issues:
  - Which end is the top?
  - Where does “top” point to?
  - What are the costs of the operations?
    - PUSH, POP, and TOP

Linked Stack

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

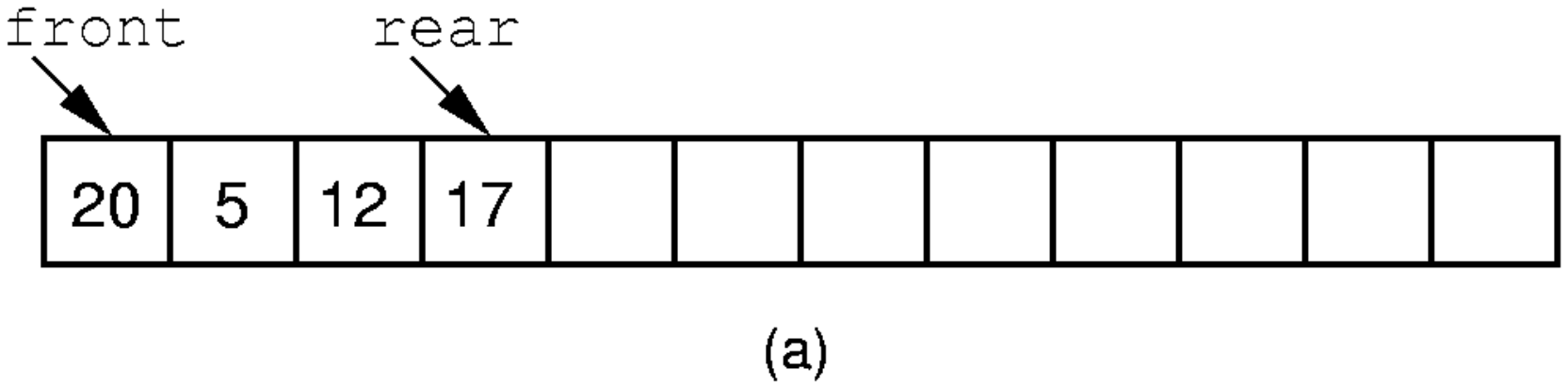
- What are the costs of the operations?

- How do space requirements compare to the array-based stack implementation?

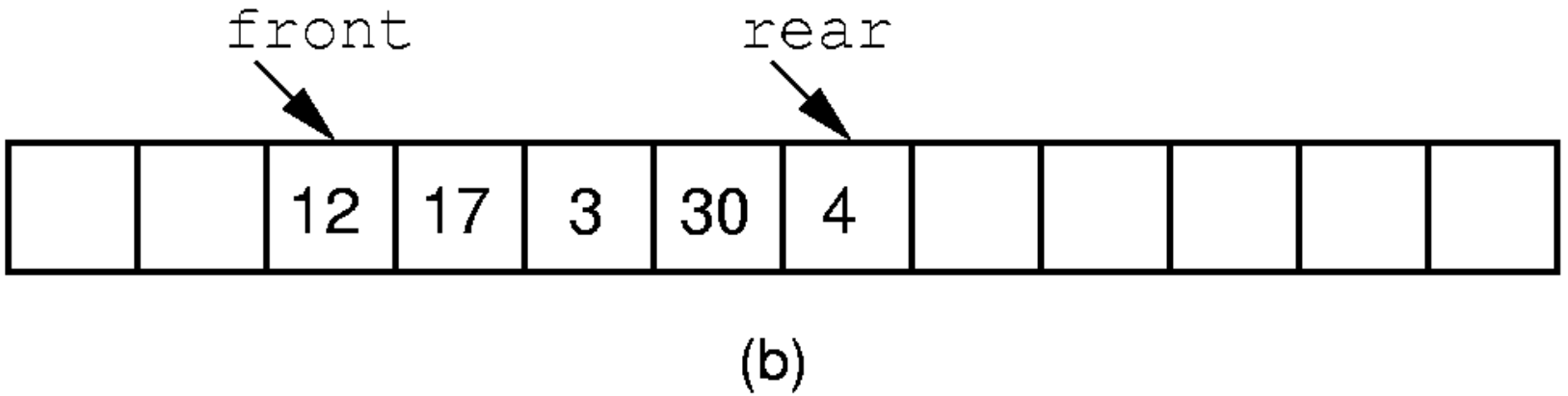
Queues

- FIFO: First in, First Out
- Restricted form of list: Insert at one end, remove from the other.
- Notation:
  - **Enqueue** for insertion
  - **Dequeue** for deletion
  - **Front** for accessing first element
  - **Rear** for accessing last element

Queue Implementation (1)

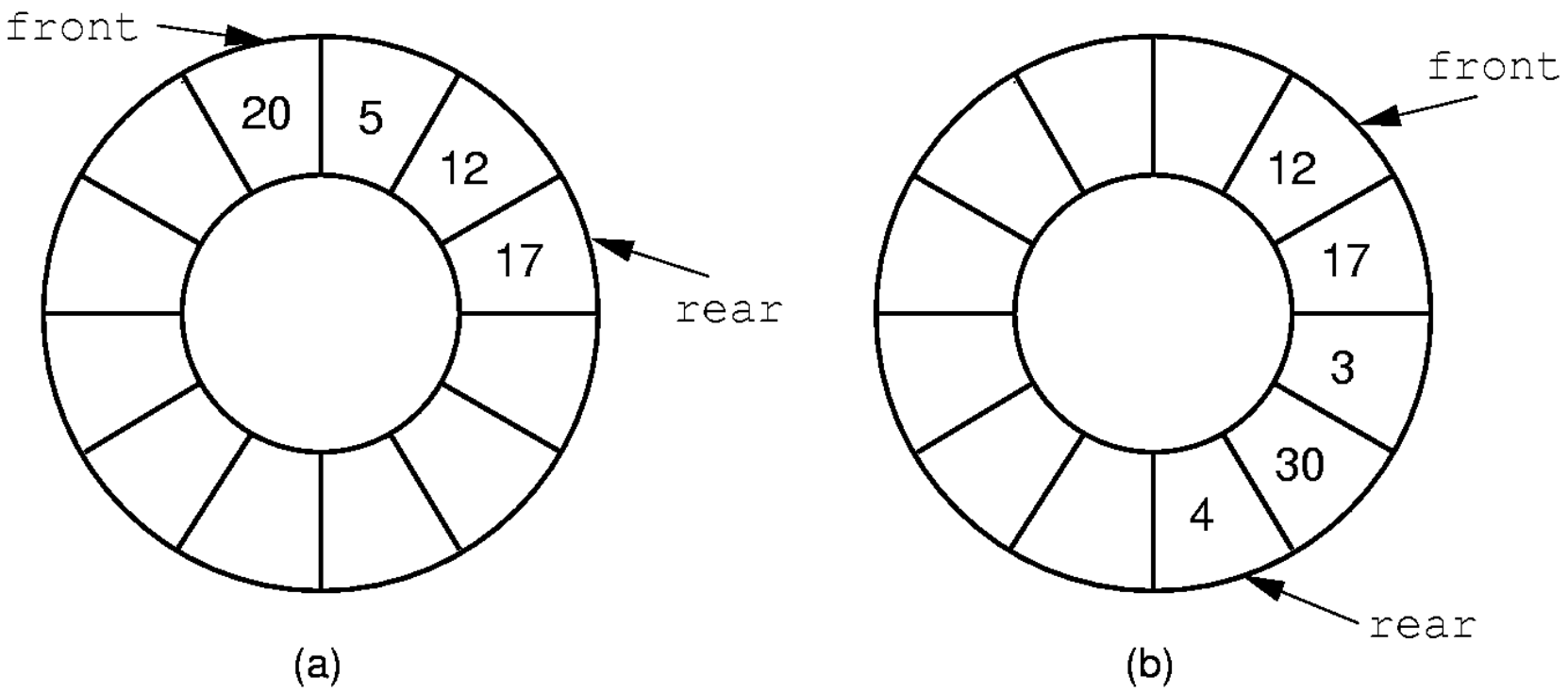


(a)



(b)

Queue Implementation (2)



"Circular" Queue

Dictionary

- Often want to insert records, delete records, and search for records.

- Required concepts:
  - Search key: describe what we are looking for
  - Key comparison
    - Equality: sequential search
    - Relative order: sorting

Records and Keys

- Problem: How do we extract the key from a record?
- Records can have multiple keys.
- Fundamentally, the key is not a property of the record, but of the context.
- Solution: We will explicitly store the key with the record.

Dictionary ADT

```
public interface Dictionary<Key, E> {

  public void clear();
  public void insert(Key k, E e);
  public E remove(Key k); // Null if none
  public E removeAny();   // Null if none
  public E find(Key k);   // Null if none
  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; }
}
```

Using Dictionary

```
// 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");
```

Unsorted List Dictionary

```
class UALdictionary<Key, E>
      implements Dictionary<Key, E> {

  private static final int defaultSize = 10;
  private AList<KVpair<Key, E>> list;

  // Constructors
  UALdictionary() { this(defaultSize); }
  UALdictionary(int sz)
    { list = new AList<KVpair<Key, E>>(sz); }

  public void clear() { list.clear(); }

  /** Insert an element: append to list */
  public void insert(Key k, E e) {
    KVpair<Key,E> temp = new KVpair<Key,E>(k, e);
    list.append(temp);
  }
```

Sorted vs. Unsorted Array List Dictionaries

- If list were sorted
  - Could use binary search to speed search
  - Would need to insert in order, slowing insert
- Which is better?
  - If lots of searches, sorted list is good
  - If insertions are much more frequent, then sorting has no benefit.

What you need to know

- Motivation and main idea of doubly linked list
  - Doubly linked list vs. singly linked list
- Stack and Queue data structure
  - Array based stack vs. link based stack
  - How to implement circular queue using array
- Dictionary data structure
  - Sorted vs. unsorted array list dictionary

Questions?