CS1102: Data Structures and Algorithms

Part 6

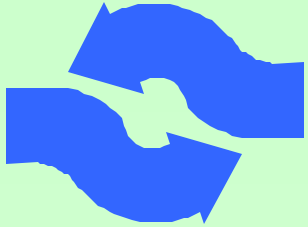
Priority Queue Table

Zoltan KATO

S16 06-12

Adopted from Chin Wei Ngan's cs1102 lecture notes

Priority Queues & Tables



- What is a PQ?
- PQ ADT
- Implementation of PQ
  - Sorted List
  - Set of Queues
- What is a Table?
- Table ADT
- Implementation of Table
  - Sorted Array

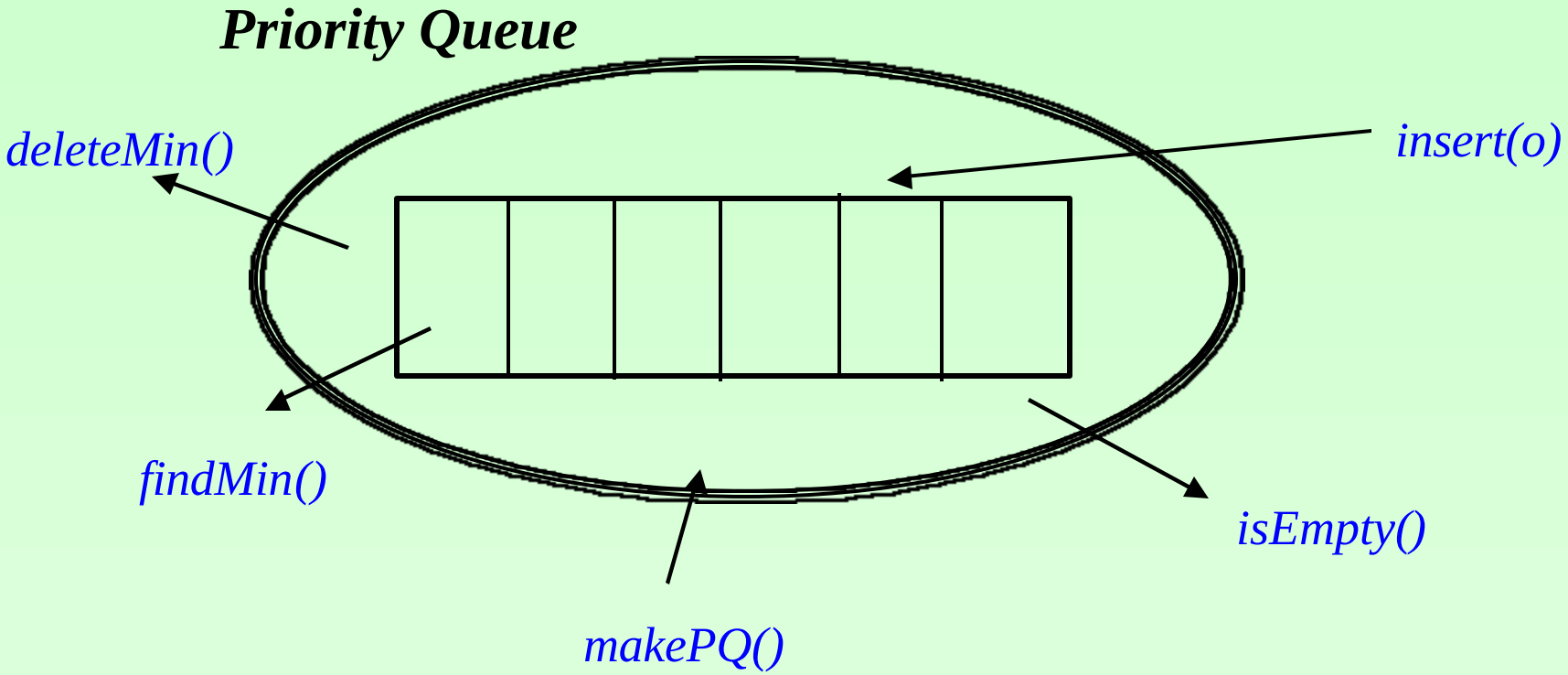
What is a Priority Queue?

Priority Queues order its elements based on *priority*, followed by *arrival sequence*.

Examples:

- Shared Printers
- MultiTasking Operating Systems
- Express Queue in Supermarkets

Priority Queue ADT



Interface Priority Queue ADT

```
interface PQueue {

 public void insert(Comparable o);

 public void deleteMin() throws Underflow;

 public Comparable findMin() throws Underflow;

 public boolean isEmpty();

}
```

Implementations for PQ

- *sorted array*
  difficult to `insert`, but very easy to `findMin`

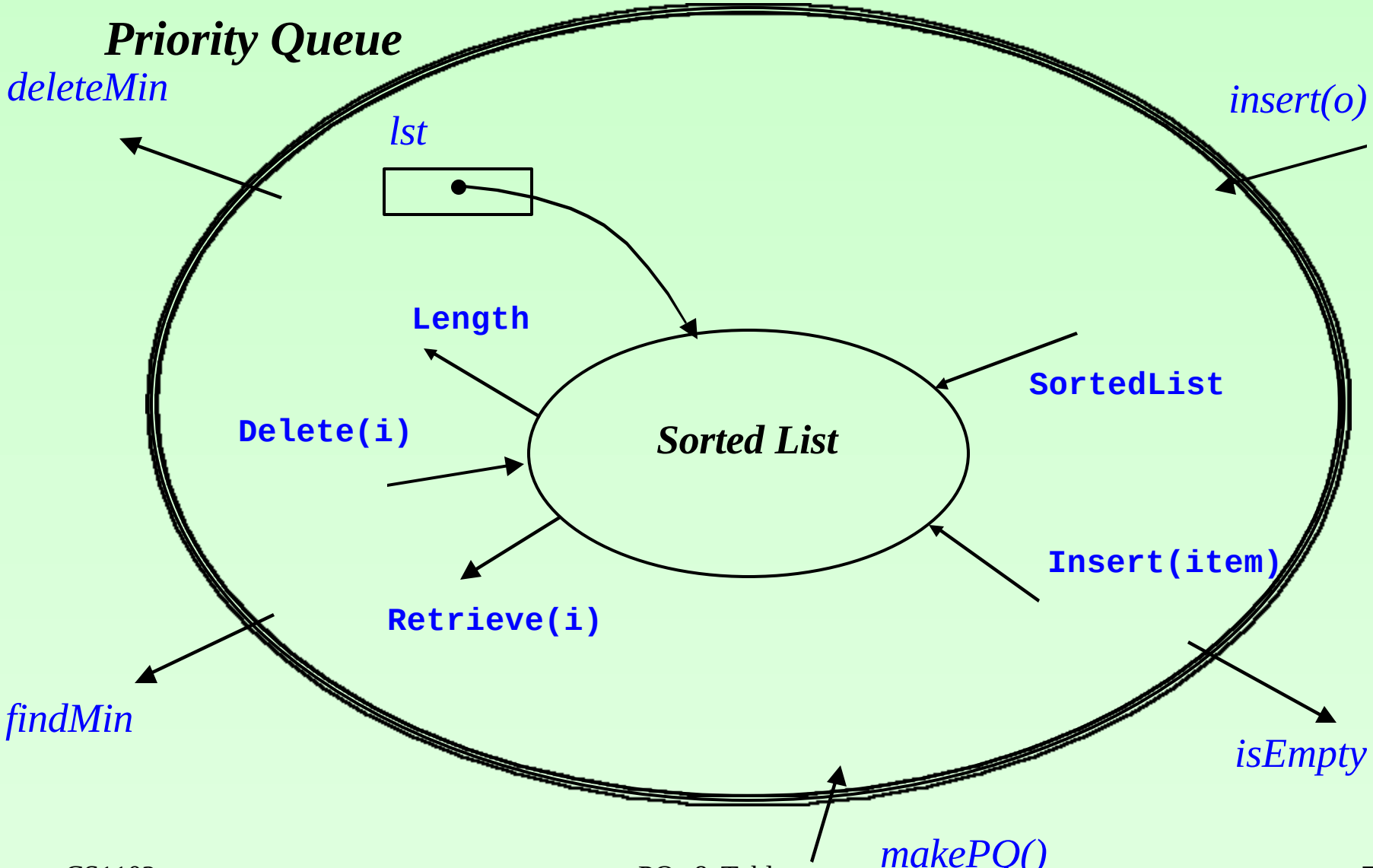
- *sorted linked-list*
  difficult to find position to `insert`, but very easy to `findMin`

- *unsorted array*
  easy to `insert`, but hard to `findMin`

- *unsorted linked-list*
  easy to `insert`, but hard to `findMin`

- *heap (tree-like) data structure (Lectures on Heap)*
  fast `insert`, `deleteMin`, and `findMin`!!

Implementation of PQ (Sorted List)



Node Implementation of PQ (Sorted List)

```
class PQNode implements Comparable{

 Object item;
 int priority;

 public PQNode(Object o, int p)
  { priority=p; item=o;
  }



public int compareTo(Object o)
  {PQNode o2 = (PQNode) o;
   if (priority < o2.priority) return –1;
   else {if (priority==o2.priority) return 0;
         else return 1;};
  }
```

Codes — Implementation of PQ (Sorted List)

```
class PQSortL implements PQueue {

 private SortedList lst;

 public PQSortL() {lst = new SortedList(); }
 public PQueue makePQ() {return new PQSortL(); }

 public boolean isEmpty ()
   {
     return (lst.Length()==0);
   }


 public void insert(PQNode o)
   { lst.Insert(o); }  // assumes that o is inserted at the
                       // first location prior to an elem with
                       // a bigger priority value
```

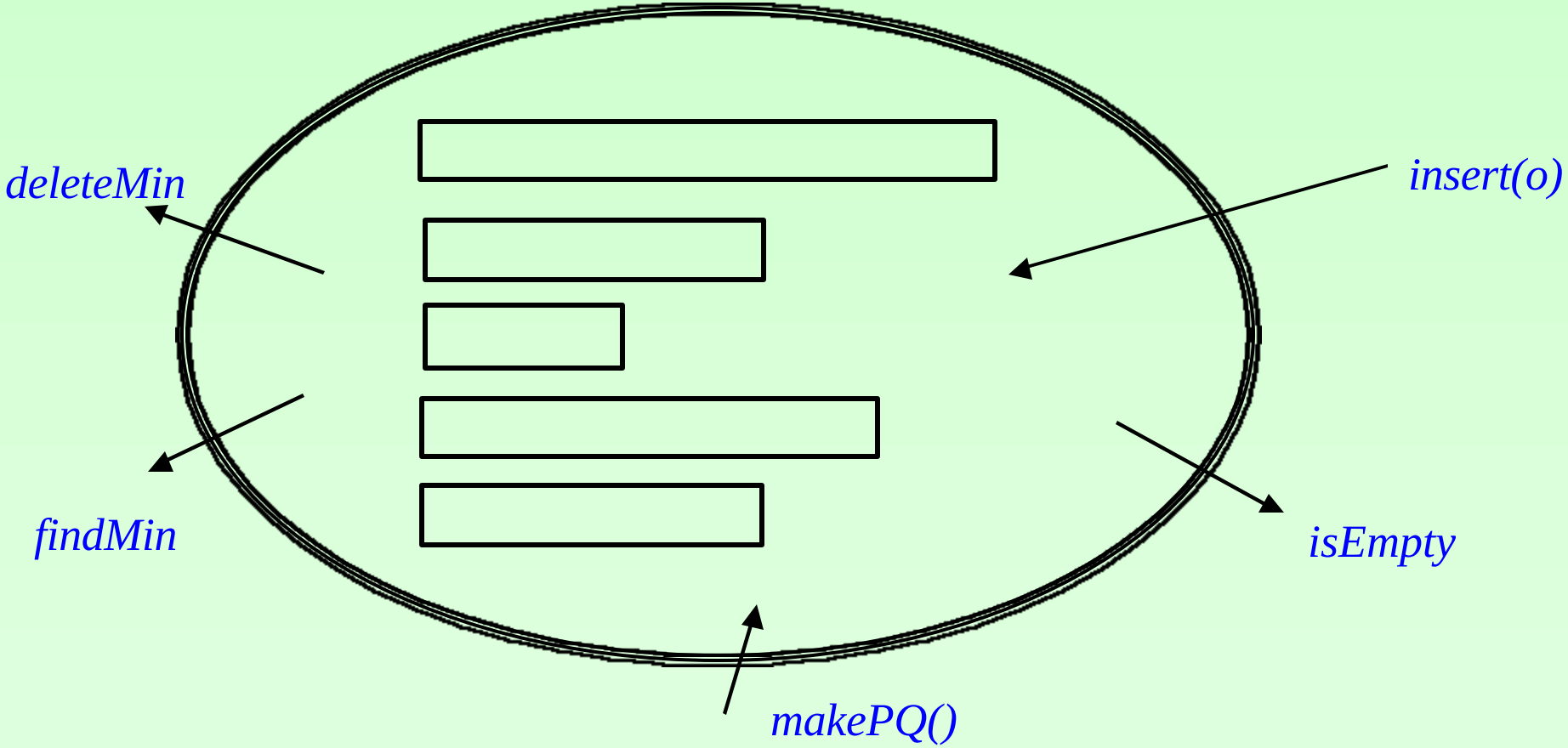
Codes — Implementation of PQ (Sorted List)

```
public Comparable findMin() throws Underflow
  {
    try {lst.Retrieve(1);
    } catch (ItemNotFound e) {
         throw new Underflow("findMin fails - empty PQ");
    };
  }

public void deleteMin() throws Underflow
  {
    try {lst.Delete(1);
    } catch (ItemNotFound e) {
         throw new Underflow("deleteMin fails - empty PQ");
    };
  }
```

Implementation of PQ (Set of Queues)

Assume priority limited to 0..4.

Priority Queue made from 5 Simpler Queues.



Node Implementation of PQ (Set of Queues)

Changes to ensure a limited set of priorities .

```
class PQNode implements Comparable{

 private int priority; // limited to 0..4
 public Object item;

 public PQNode(int p, Object o)
  {if (p<0 || p>4) {throw new Exception("Unacceptable priority")}
   else {priority=p; item=o}
  }
 public int priority()
  {return priority}

 public void setPriority(int p)
  {if (p<0 || p>4) {throw new Exception("Unacceptable priority")}
   else {priority=p}}
```

Codes Implementation of PQ (Set of Queues)

```
class PSetQ implements PQueue {

 private Queue[5] Q;

 public PQueue() {for (j=0; j<5; j++)
                    { Q[j]= QueueArr.makeQueue(); }; };
 public static makePQ() {return new PQueue();}

 public boolean isEmpty ()
   { boolean emptyflag=true; int j=0;
     while (emptyflag && j<5) do
      {if (!Q[j].isEmpty()){emptyflag=false;}
       else j++;}
     return emptyflag;
   }

 public void insert(Comparable o)
   { PQNode pn= (PQNode) o;
     Q[pn.priority()].enqueue(pn) }
```

Codes Implementation of PQ (Set of Queues)

```
public Comparable findMin() throws Underflow;
  { boolean emptyflag=true; int j=0;
    while (emptyflag && j<5) do
     { try {p = Q[j].getFront(); emptyflag=false;
       } catch (Underflow e) {j++;};
     };
    if emptyflag throw new Underflow("findMin fails - empty PQ");
    else return p;
  }


public void deleteMin() throws Underflow;
  { boolean emptyflag=true; int j=0;
    while (emptyflag && j<5) do
     { try {p = Q[j].dequeue(); emptyflag=false;
       } catch (Underflow e) {j++;};
     };
    if emptyflag throw new Underflow("deleteMin fails - empty PQ");
  }
```

What is a Tables?

Tables are used to keep a “key-to-value” mapping, where accesses are performed via keys rather than positions.

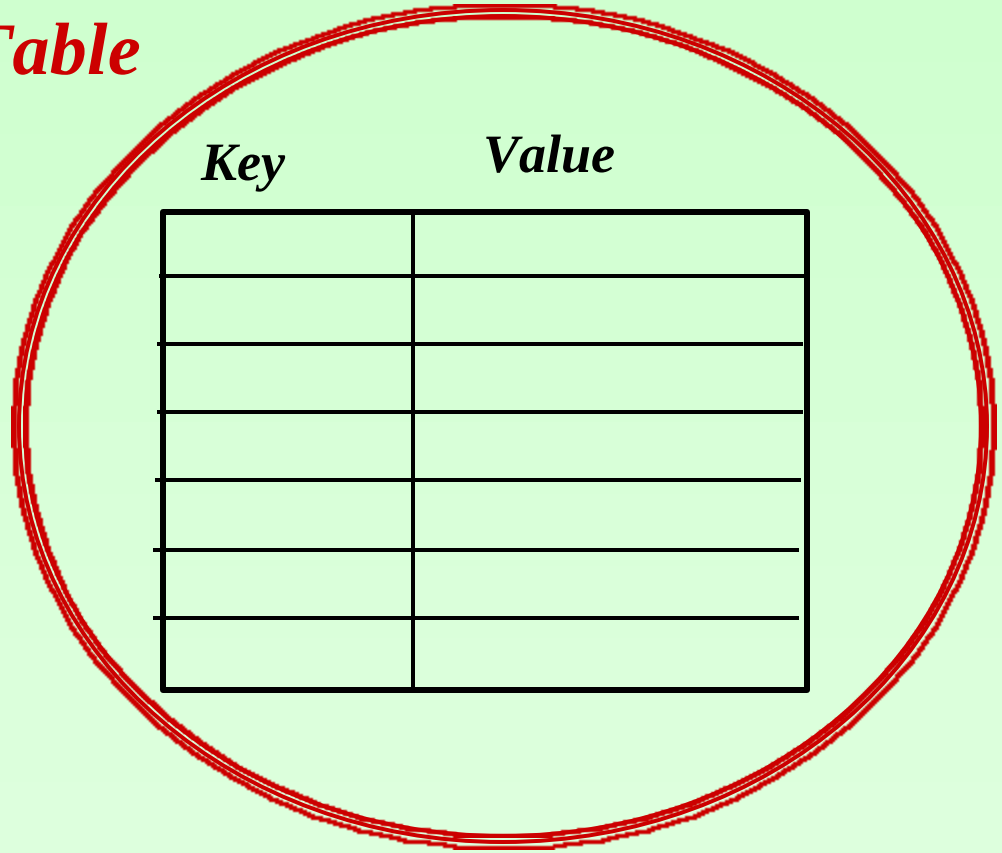
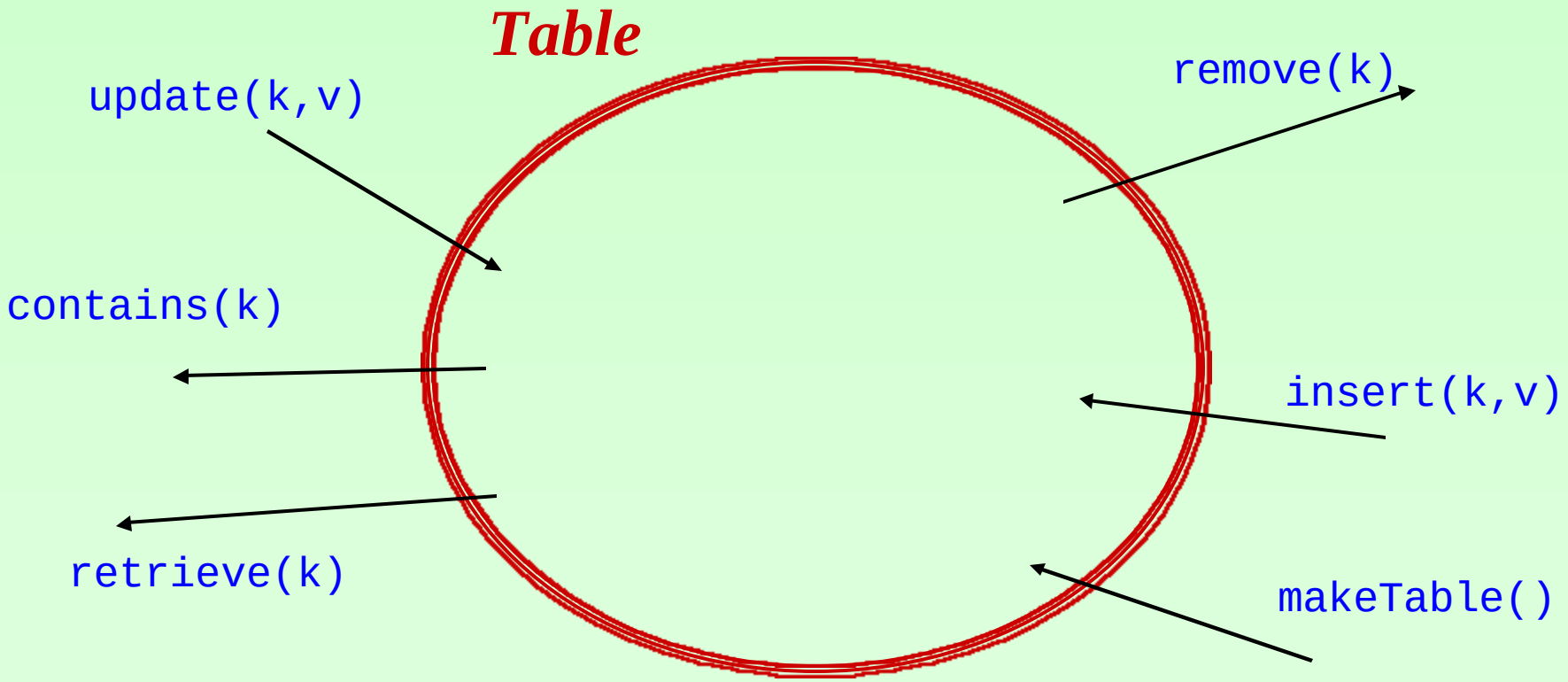


Table ADT

Table operations access elements through the key values.



Interface Table ADT

```
interface Table {

 // Table makeTable() ; static method in class

 boolean isEmpty() ;

 int size() ;

 boolean contains(Comparable key) ;

 Object retrieve(Comparable key) throws ItemNotFound;

 void insert(Comparable key, Object o); throws ItemExist;

 void remove(Comparable key) throws ItemNotFound;

 void update(Comparable key, Object o) throws ItemNotFound;

}
```

Implementations for Tables

- *unsorted array*
  easy to `insert`, but hard to `retrieve & delete`

- *unsorted linked-list*
  easy to `insert`, but hard to `retrieve & delete`

- *sorted linked-list*
  difficult to find position to `insert`, and similarly for `retrieve`

- *sorted array*
  difficult to `insert & delete`, but very easy to `retrieve`

- *binary search trees (Lecture 18)*
  fast `insert`, `retrieve` , `remove` and `update`!!

Nodes — Implementation of Tables

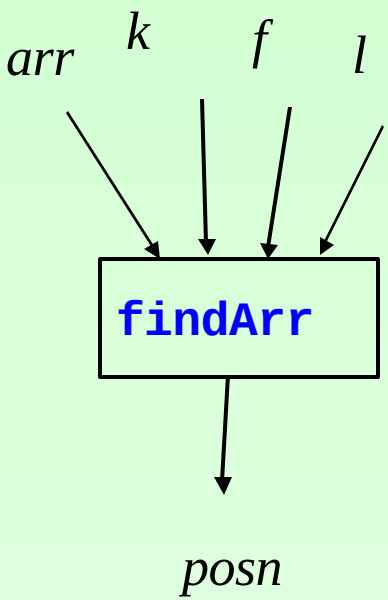
```
class TEntry implements Comparable{

 public Comparable key;
 public Object obj;

 public TEntry(Comparable k, Object o)
       {key=k; obj=o}


 public int compareTo(Object o)
       {Tentry nxt = (Tentry) o;
        if (key.lessThan(nxt.key)) {return –1;}
        else {if (key.equals(nxt.key)) {return 0;}
              else {return 1;}}
       }
}
```

Codes Implementation of Tables (Sorted Array)

```
class TableArr implements Table {

 private TEntry[] arr;
 private int last;
 private int maxSize;
 private final int initSize = 1000;
 private final int increment = 1000;

 public TableArr() {arr = new TEntry[initSize]; last = -1;
                    maxSize=initSize; }
 public static TableArr makeTableArr()
                   {return new TableArr();}

 public boolean isEmpty() { return (last<0);  }

 private boolean isFull()
      { return (last>=(maxSize-1));  }

 public int size() { return (last+1); }
```

allows resizing of table

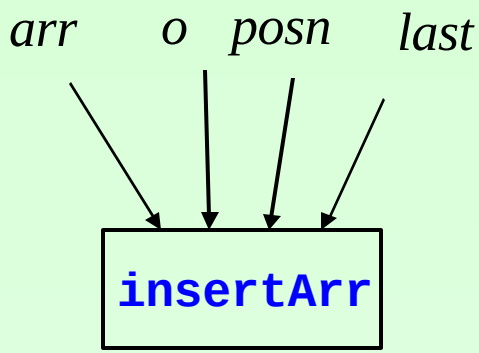
```
// search sorted arr for key k between positions
// f and l &  return position where k can be inserted
public static int findArr
  (TEntry [] arr, Comparable k, int f, int l) { … }
```

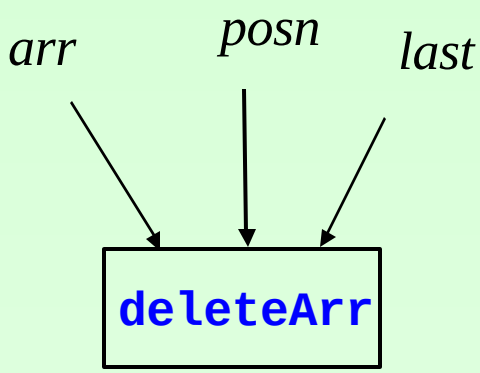


Codes (Digression) Implementation of Tables

```
// insert object o into array arr at position p
 public static void insertArr
   (TEntry[] arr, TEntry o, int posn, int last) { … }
```

```
// delete object of array arr at position p
 public static void deleteArr
   (TEntry[] arr, int posn, int last) { … }
```





Codes Implementation of Tables (Sorted Array)

```
public void insert(Comparable k, Object o) throws ItemExist
  { if (this.isFull()) { this.enlargeArr(); };
    int p=findArr(arr,k,0,last);
    if (p<=last && arr[p].key.equals(k))
         {throw new ItemExist("insert fails");}
    else {insertArr(arr,TEntry(k,o),p,last); last++;};
  }
```

```
public boolean contains(Comparable k)
  {int p=findArr(arr,k,0,last);
   return (p<=last && k.equals(arr[p].key));
  }
```

Codes Implementation of Tables (Sorted Array)

```
public void remove(Comparable k) throws ItemNotFound
  { int p=findArr(arr,k,0,last);
    if (p>last || !arr[p].key.equals(k))
         throw new ItemNotFound("remove fails");
    else {deleteArr(arr,p,last); last--;};
  }
public Object retrieve(Comparable k) throws ItemNotFound
  {int p=findArr(arr,k,0,last);
   if (p>last || !arr[p].key.equals(k))
         throw new ItemNotFound("retrieve fails");
   else return arr[p].obj;
  }
public void update(Comparable k, Object o) throws ItemNotFound
  {int p=findArr(arr,k,0,last);
   if (p>last || !arr[p].key.equals(k))
         throw new ItemNotFound("update fails");
   else arr[p].obj=o;
  }
```