Algorithms

(Arrays and Lists)

Pramod Ganapathi
Department of Computer Science
State University of New York at Stony Brook

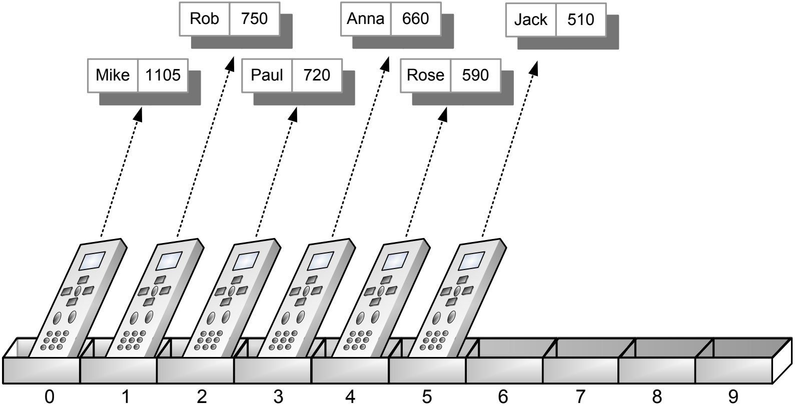
December 29, 2020

Contents

- Implementations
  - Arrays
  - Singly Linked Lists
  - Circularly Linked Lists
  - Doubly Linked Lists
- Abstract Data Types
  - Array Lists
  - Positional Lists

Arrays

Scoreboard: Storing game entries in an array



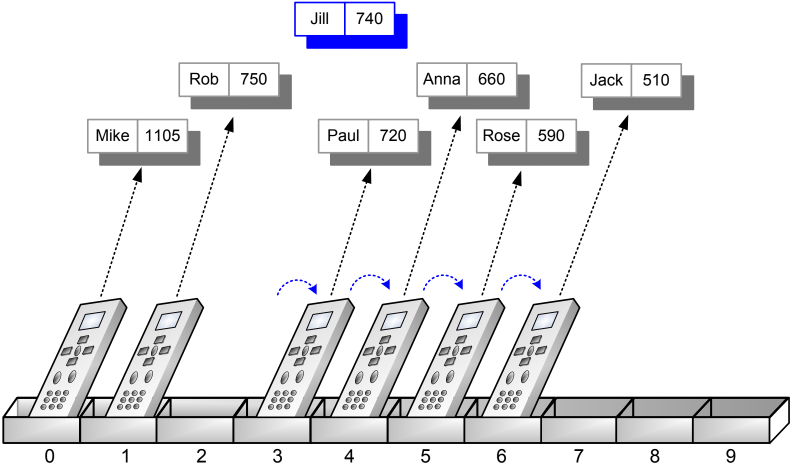
Scoreboard: High score entry

```java
public class GameEntry {
  private String name;   // name of the person earning this score
  private int score;     // the score value

  /** Constructs a game entry with given parameters.. */
  public GameEntry(String n, int s) { name = n; score = s; }
  /** Returns the name field. */
  public String getName() { return name; }
  /** Returns the score field. */
  public int getScore() { return score; }
  /** Returns a string representation of this entry. */
  public String toString() { return "(" + name + ", " + score + ")"; }
}
```

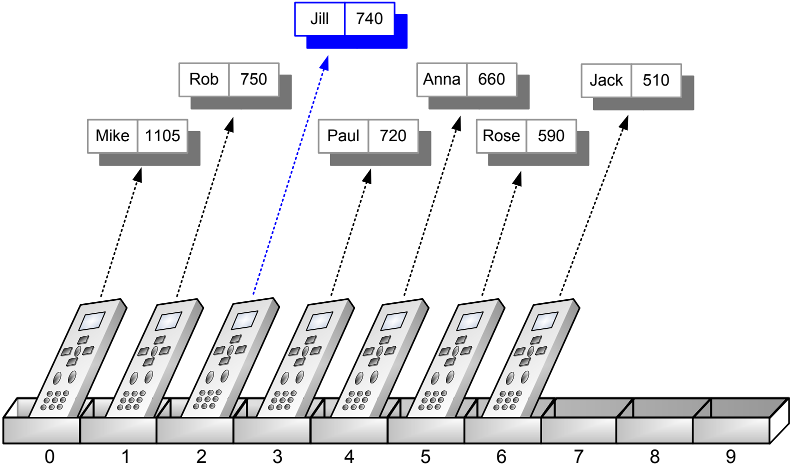
Scoreboard: Board of high scores

```java
/** Class for storing high scores in an array in nondecreasing order. */
public class Scoreboard {
  private int numEntries = 0;                // number of actual entries
  private GameEntry[] board;                 // array of game entries

  /** Constructs an empty scoreboard with the given capacity. */
  public Scoreboard(int capacity) { board = new GameEntry[capacity]; }
  /** Attempt to add a new high score to the collection. */
  public void add(GameEntry e) {...}
  /** Remove and return the high score at index i. */
  public GameEntry remove(int i) throws IndexOutOfBoundsException {...}
  /** Returns a string representation of the high scores list. */
  public String toString() {...}

  public static void main(String[] args) {...}
}
```

Scoreboard: Add an entry



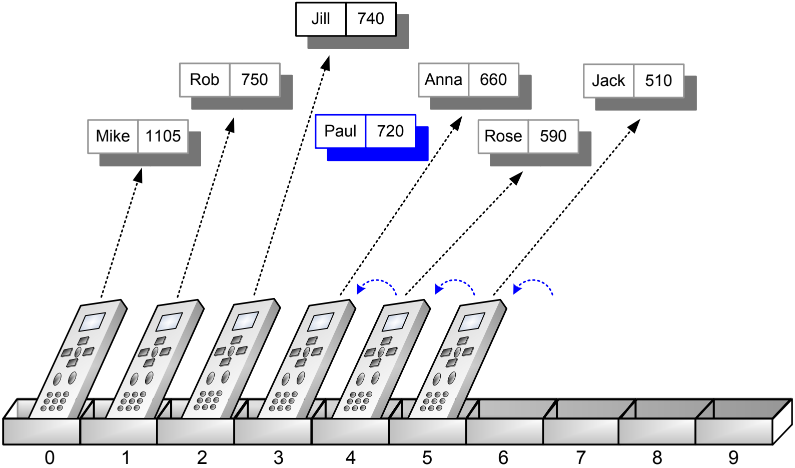
Scoreboard: Add an entry



Scoreboard: Add an entry

```java
/** Attempt to add a new score to the collection */
public void add(GameEntry e) {
  int newScore = e.getScore();

  // is the new entry e really a high score?
  if (numEntries < board.length || newScore > board[numEntries-1].getScore()) {
    if (numEntries < board.length)        // no score drops from the board
      numEntries++;                       // so overall number increases
    // shift any lower scores rightward to make room for the new entry
    int j = numEntries - 1;
    while (j > 0 && board[j-1].getScore() < newScore) {
      board[j] = board[j-1];              // shift entry from j-1 to j
      j--;                                // and decrement j
    }
    board[j] = e;                         // when done, add new entry
  }
}
```

Scoreboard: Remove an entry



Scoreboard: Remove an entry

```java
/** Remove and return the high score at index i. */
public GameEntry remove(int i) throws IndexOutOfBoundsException {
  if (i < 0 || i >= numEntries)
    throw new IndexOutOfBoundsException("Invalid index: " + i);
  GameEntry temp = board[i];                   // save the object to be removed
  for (int j = i; j < numEntries - 1; j++)    // count up from i (not down)
    board[j] = board[j+1];                     // move one cell to the left
  board[numEntries - 1] = null;                // null out the old last score
  numEntries--;
  return temp;                                 // return the removed object
}
```

Scoreboard: toString function

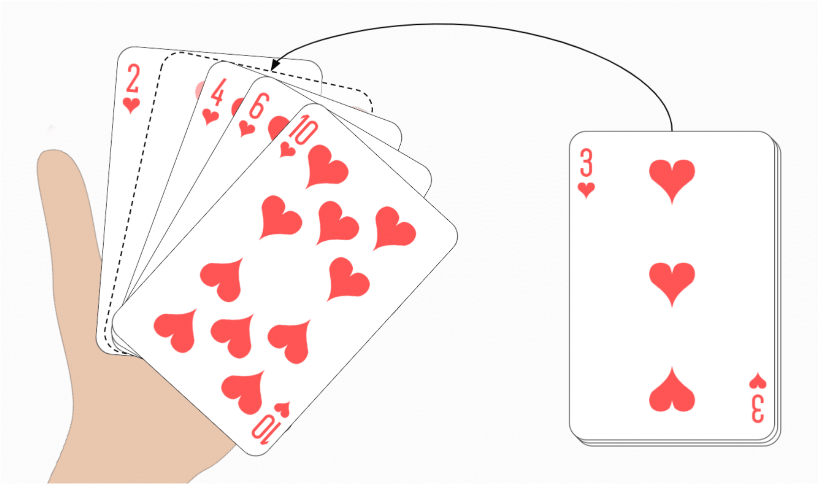
- Print the board consisting of high scores:
  `[board[0], board[1], ..., board[numEntries-1]]`

```java
/** Returns a string representation of the high scores list. */
public String toString() {
  StringBuilder sb = new StringBuilder("[");

  for (int j = 0; j < numEntries; j++) {
    if (j > 0) sb.append(", ");                    // separate entries by commas
    sb.append(board[j]);
  }
  sb.append("]");
  return sb.toString();
}
```

Scoreboard: main function

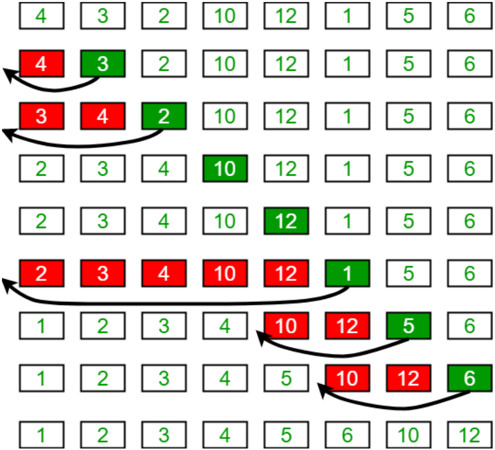
```java
public static void main(String[] args) {
  Scoreboard highscores = new Scoreboard(5);
  String[] names = {"Rob","Mike","Rose","Jill","Jack","Anna","Paul","Bob"};
  int[] scores = {750, 1105, 590, 740, 510, 660, 720, 400};

  for (int i = 0; i < names.length; i++) {
    GameEntry ge = new GameEntry(names[i], scores[i]);
    System.out.println("Adding " + ge);
    highscores.add(ge);
    System.out.println(" Scoreboard: " + highscores);
  }
  System.out.println("Remove score at index " + 3); highscores.remove(3);
  System.out.println(highscores);
  System.out.println("Remove score at index " + 0); highscores.remove(0);
  System.out.println(highscores);
  System.out.println("Remove score at index " + 1); highscores.remove(1);
  System.out.println(highscores);
  System.out.println("Remove score at index " + 1); highscores.remove(1);
  System.out.println(highscores);
  System.out.println("Remove score at index " + 0); highscores.remove(0);
  System.out.println(highscores);
}
```

Sorting: Insertion sort



https://www.happycoders.eu/wp-content/uploads/2020/05/Insertion_Sort_Playing_Card_Example.png

Sorting: Insertion sort



4 3 2 10 12 1 5 6

4 3 2 10 12 1 5 6

3 4 2 10 12 1 5 6

2 3 4 10 12 1 5 6

2 3 4 10 12 1 5 6

2 3 4 10 12 1 5 6

1 2 3 4 10 12 5 6

1 2 3 4 5 10 12 6

1 2 3 4 5 6 10 12

Source: `https://media.geeksforgeeks.org/wp-content/uploads/insertionsort.png`

Sorting: Insertion sort

```
INSERTION-SORT($A[0..n-1]$)
Input: An array $A[0..n-1]$ of $n$ orderable elements
Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 1$ to $n-1$ do
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \geq 0$ and $A[j] > v$ do
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$
```

Built-in methods for java.util.Arrays class

Method	Functionality
`equals(A, B)`	Compares arrays A and B.
`fill(A, x)`	Stores x in every cell of array A.
`copyOf(A, n)`	Returns n-sized array where the first $k = \min\{n, A.\text{length}\}$ elements are copied from A. If $n > A.\text{length}$, then the remaining elements are padded with 0 or null.
`toString(A)`	Returns string representation of array A.
`sort(A)`	Sorts array A based on natural ordering.
`binarySearch(A)`	Searches the sorted array A for value x.

Pseudorandom numbers

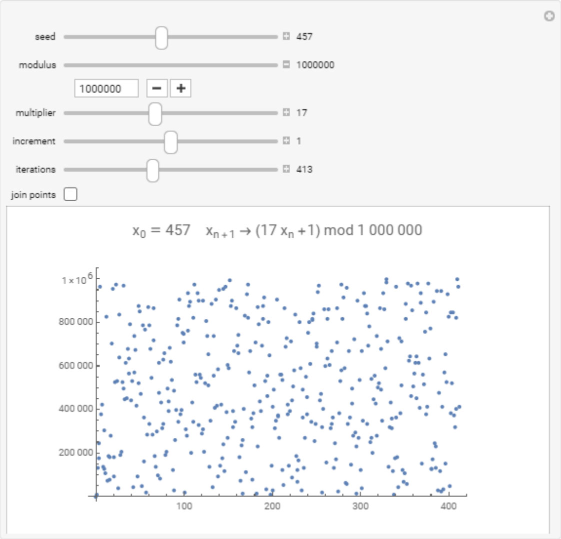
Linear congruential generator

$$X_i = \begin{cases} \text{seed} & \text{if } i = 0 \\ (a \times X_{i-1} + b) \ \% \ n & \text{if } i \geq 1 \end{cases}$$

Example

- Suppose seed $= 467$, $a = 17$, $b = 1$, $n = 1$ million. Then
  $X_0 = 467$
  $X_1 = (17 \times 467 + 1) \ \% \ 10^6 = 7940$
  $X_2 = (17 \times 7940 + 1) \ \% \ 10^6 = 134981$
  $X_3 = (17 \times 134981 + 1) \ \% \ 10^6 = 294678$
  $X_4 = (17 \times 294678 + 1) \ \% \ 10^6 = 9527$
  $X_5 = (17 \times 9527 + 1) \ \% \ 10^6 = 161960$
  $X_6 = (17 \times 161960 + 1) \ \% \ 10^6 = 753321$
  $X_7 = (17 \times 753321 + 1) \ \% \ 10^6 = 806458$

Pseudorandom numbers



seed 457

modulus 1000000

1000000 − +

multiplier 17

increment 1

iterations 413

join points □

$x_0 = 457 \quad x_{n+1} \to (17\, x_n + 1) \bmod 1\,000\,000$

https://demonstrations.wolfram.com/LinearCongruentialGenerators/
https://asecuritysite.com/encryption/linear

Built-in methods for java.util.Random class

Method	Functionality
`nextBoolean()`	Returns the next pseudorandom boolean value.
`nextDouble()`	Returns the next pseudorandom double value in the range $[0.0, 1.0]$
`nextInt()`	Returns the next pseudorandom int.
`nextInt(n)`	Returns the next pseudorandom int in the range $[0, n)$.
`setSeed(s)`	Sets the seed of the generator to the long s.

2-D Arrays

- Definition.

  A 2-D array in Java is created as array of arrays

- Declaration.

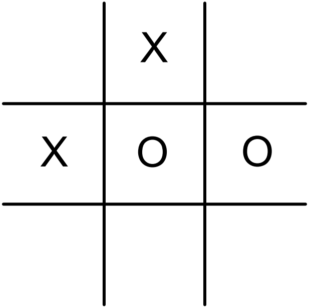
  ```
  int[][] data = new int[8][10];
  ```

- Valid uses.

  ```
  data[i][i+1] = data[i][i] + 3;
  j = data.length;        // j is 8
  k = data[4].length;     // k is 10
  ```

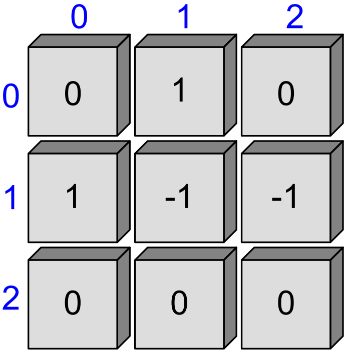
	0	1	2	3	4	5	6	7	8	9
0	22	18	709	5	33	10	4	56	82	440
1	45	32	830	120	750	660	13	77	20	105
2	4	880	45	66	61	28	650	7	510	67
3	940	12	36	3	20	100	306	590	0	500
4	50	65	42	49	88	25	70	126	83	288
5	398	233	5	83	59	232	49	8	365	90
6	33	58	632	87	94	5	59	204	120	829
7	62	394	3	4	102	140	183	390	16	26

2-D Arrays: Tic-Tac-Toe



	X	
X	O	O

playing board



	0	1	2
0	0	1	0
1	1	-1	-1
2	0	0	0

board array

2-D Arrays: Tic-Tac-Toe

```java
/** Simulation of a Tic-Tac-Toe game (does not do strategy). */
public class TicTacToe {
  public static final int X = 1, O = -1;      // players
  public static final int EMPTY = 0;          // empty cell
  private int board[][] = new int[3][3];      // game board
  private int player;                         // current player

  /** Constructor */
  public TicTacToe() { clearBoard(); }
  /** Clears the board */
  public void clearBoard() {...}
  /** Puts an X or O mark at position i,j. */
  public void putMark(int i, int j) throws IllegalArgumentException {...}
  /** Checks whether the board configuration is a win for the given player. */
  public boolean isWin(int mark) {...}
  /** Returns the winning player's code, or 0 to indicate a tie.*/
  public int winner() {...}
  /** Returns a simple character string showing the current board. */
  public String toString() {...}
  /** Test run of a simple game */
  public static void main(String[] args) {...}
}
```

2-D Arrays: Tic-Tac-Toe

```
/** Clears the board */
public void clearBoard() {
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      board[i][j] = EMPTY;                    // every cell should be empty
  player = X;                                 // the first player is 'X'
}
```

```
/** Puts an X or O mark at position i,j. */
public void putMark(int i, int j) throws IllegalArgumentException {
  if ((i < 0) || (i > 2) || (j < 0) || (j > 2))
    throw new IllegalArgumentException("Invalid board position");
  if (board[i][j] != EMPTY)
    throw new IllegalArgumentException("Board position occupied");
  board[i][j] = player;       // place the mark for the current player
  player = - player;          // switch players (uses fact that O = - X)
}
```

2-D Arrays: Tic-Tac-Toe

```java
/** Checks whether the board configuration is a win for the given player. */
public boolean isWin(int mark) {
  return ((board[0][0] + board[0][1] + board[0][2] == mark*3)   // row 0
       || (board[1][0] + board[1][1] + board[1][2] == mark*3)   // row 1
       || (board[2][0] + board[2][1] + board[2][2] == mark*3)   // row 2
       || (board[0][0] + board[1][0] + board[2][0] == mark*3)   // column 0
       || (board[0][1] + board[1][1] + board[2][1] == mark*3)   // column 1
       || (board[0][2] + board[1][2] + board[2][2] == mark*3)   // column 2
       || (board[0][0] + board[1][1] + board[2][2] == mark*3)   // diagonal
       || (board[2][0] + board[1][1] + board[0][2] == mark*3)); // rev diag
}
```

```java
/** Returns the winning player's code, or 0 to indicate a tie.*/
public int winner() {
  if (isWin(X))        return(X);
  else if (isWin(O))   return(O);
  else                 return(0);
}
```

2-D Arrays: Tic-Tac-Toe

```
/** Returns a simple character string showing the current board. */
public String toString() {
  StringBuilder sb = new StringBuilder();
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
      switch (board[i][j]) {
        case X:     sb.append("X"); break;
        case O:     sb.append("O"); break;
        case EMPTY: sb.append(" "); break;
      }
      if (j < 2) sb.append("|");                 // column boundary
    }
    if (i < 2) sb.append("\n-----\n");           // row boundary
  }
  return sb.toString();
}
```

2-D Arrays: Tic-Tac-Toe

```java
/** Test run of a simple game */
public static void main(String[] args) {
  TicTacToe game = new TicTacToe();
  /* X moves: */          /* O moves: */
  game.putMark(1,1);      game.putMark(0,2);
  game.putMark(2,2);      game.putMark(0,0);
  game.putMark(0,1);      game.putMark(2,1);
  game.putMark(1,2);      game.putMark(1,0);
  game.putMark(2,0);
  System.out.println(game);
  int winningPlayer = game.winner();
  String[] outcome = {"O wins", "Tie", "X wins"};  // rely on ordering
  System.out.println(outcome[1 + winningPlayer]);
}
```

```
O|X|O
-----
O|X|X
-----
X|O|X
Tie
```

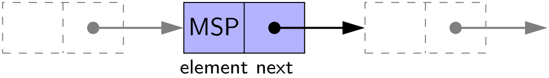
Advantages and disadvantages

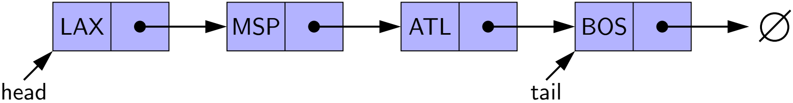
Operation	Time complexity
Fast operations	
Access/modify	$\mathcal{O}\left(1\right)$
Insert last	$\mathcal{O}\left(1\right)$
Slow operations	
Insert	$\mathcal{O}\left(n\right)$
Delete	$\mathcal{O}\left(n\right)$
Increase size	–

Singly Linked Lists

Singly linked lists

- A singly linked list, an alternative of array, is a linear sequence of nodes.
- E.g.: A singly linked list of airport codes.





Node class

```
//---------------- nested Node class ----------------
/** Node of a singly linked list, which stores a reference to its
    element and to the subsequent node in the list (or null if this
    is the last node). */
private static class Node<E> {
  private E element;        // reference to the element stored at this node
  private Node<E> next;     // reference to the subsequent node in the list

  /** Creates a node with the given element and next node. */
  public Node(E e, Node<E> n) { element = e; next = n; }
  /** Returns the element. */
  public E getElement() { return element; }
  /** Returns the node that follows this one (or null if no such node). */
  public Node<E> getNext() { return next; }
  /** Sets the node's next reference to point to Node n. */
  public void setNext(Node<E> n) { next = n; }
} //----------- end of nested Node class -----------
```

SinglyLinkedList class

Method	Functionality
`size()`	Returns the number of elements in the list.
`isEmpty()`	Returns true if the list is empty, and false otherwise.
`first()`	Returns the first element in the list.
`last()`	Returns the last element in the list.
`addFirst(e)`	Adds a new element to the front of the list.
`addLast(e)`	Adds a new element to the end of the list.
`removeFirst()`	Removes and returns the first element of the list.

SinglyLinkedList class

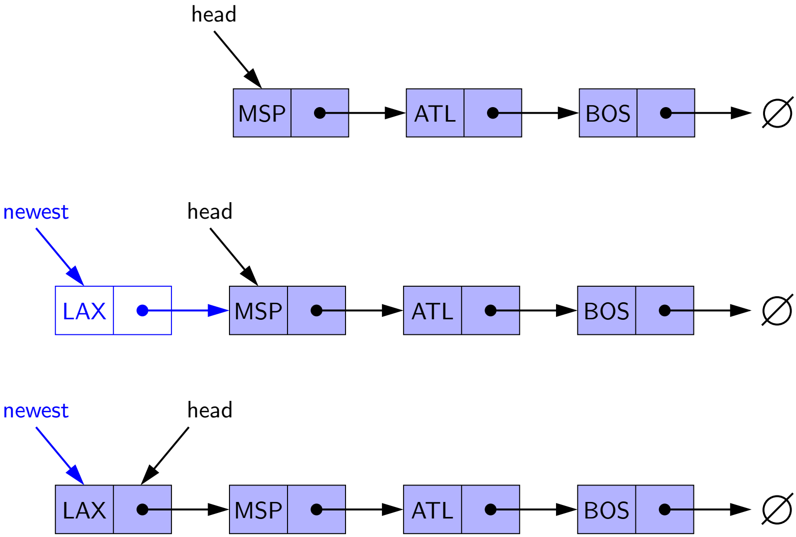
```java
public class SinglyLinkedList<E> {
  private static class Node<E> {...}

  private Node<E> head = null;     // head node of the list
  private Node<E> tail = null;     // last node of the list
  private int size = 0;            // number of nodes in the list

  public SinglyLinkedList() { }    // constructs an initially empty list

  // access methods
  public int size() { return size; }
  public boolean isEmpty() { return size == 0; }
  public E first() {...}           // returns the first element
  public E last() {...}            // returns the last element

  // update methods
  public void addFirst(E e) {...}  // adds element e to the front of the list
  public void addLast(E e) {...}   // adds element e to the end of the list
  public E removeFirst() {...}     // removes and returns the first element
}
```

Head and the tail

```java
public E first() {      // returns (but does not remove) the first element
  if (isEmpty()) return null;
  return head.getElement();
}
```

```java
public E last() {       // returns (but does not remove) the last element
  if (isEmpty()) return null;
  return tail.getElement();
}
```

Insert an element at the head

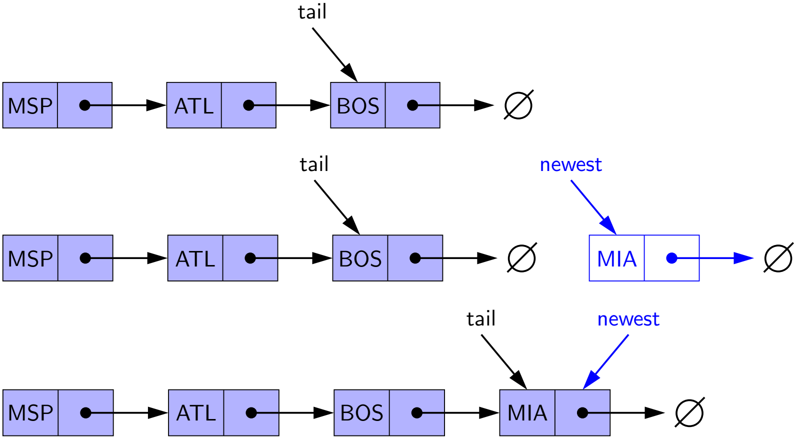


Insert an element at the head

```
ADD-FIRST($e$)
$newest \leftarrow \textsc{Node}(e)$
$newest.next \leftarrow head$
$head \leftarrow newest$
$size \leftarrow size + 1$
```

```java
public void addFirst(E e) {    // adds element e to the front of the list
  head = new Node<>(e, head);  // create and link a new node
  if (size == 0)
    tail = head;               // special case: new node becomes tail also
  size++;
}
```

Insert an element at the tail

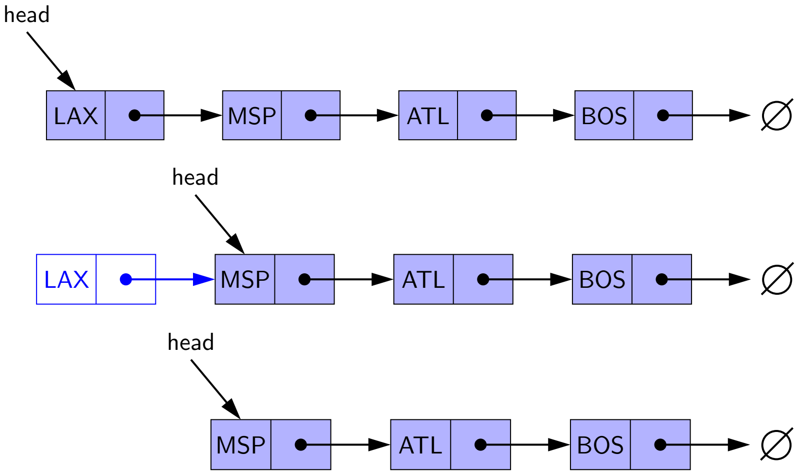


Insert an element at the tail

```
ADD-LAST($e$)
$newest \leftarrow \text{NODE}(e)$
$newest.next \leftarrow null$
$tail.next \leftarrow newest$
$tail \leftarrow newest$
$size \leftarrow size + 1$
```

```java
public void addLast(E e) {                       // adds element e to the end of the list
  Node<E> newest = new Node<>(e, null); // node will eventually be the tail
  if (isEmpty())
    head = newest;                      // special case: previously empty list
  else
    tail.setNext(newest);               // new node after existing tail
  tail = newest;                        // new node becomes the tail
  size++;
}
```

Remove an element at the head



Remove an element at the head

```
REMOVE-FIRST()
if $head = null$ then
    the list is empty
$head \leftarrow head.next$
$size \leftarrow size - 1$
```

```
public E removeFirst() {          // removes and returns the first element
  if (isEmpty()) return null;     // nothing to remove
  E answer = head.getElement();
  head = head.getNext();          // will become null if list had only one node
  size--;
  if (size == 0)
    tail = null;                  // special case as list is now empty
  return answer;
}
```

Advantages and disadvantages

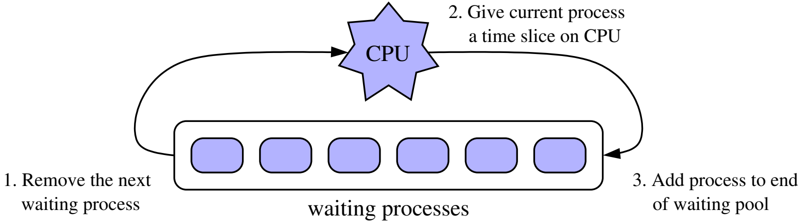
Operation	Time complexity
Fast operations	
Insert first	$\mathcal{O}(1)$
Insert last	$\mathcal{O}(1)$
Delete first	$\mathcal{O}(1)$
Increase size	$\mathcal{O}(1)$
Slow operations	
Delete last	$\mathcal{O}(n)$
Access/modify	$\mathcal{O}(n)$
Insert	$\mathcal{O}(n)$
Delete	$\mathcal{O}(n)$

Circularly Linked Lists

Applications requiring cyclic order

- Operating system
  Round-robin scheduling of processes/jobs
- Multiplayer games
  Scheduling of player turns
- Buses and subways
  Scheduling of stops in a continuous loop

Round-robin scheduling of processes

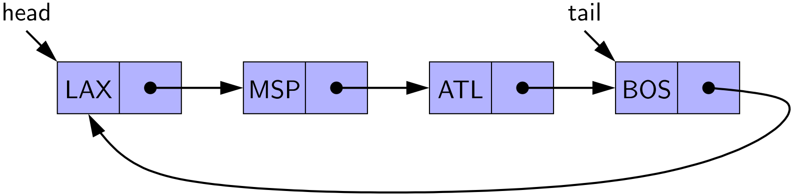

CPU
2. Give current process a time slice on CPU
1. Remove the next waiting process
waiting processes
3. Add process to end of waiting pool

Round-robin scheduler can be implemented using a singly linked list L by repeatedly performing:

1. process $p = L$.removeFirst()
2. Give a time slice to process p
3. L.addLast(p)

Designing a CircularlyLinkedList class

CircularlyLinkedList $=$
SingularlyLinkedList $+$ (tail.next $\leftarrow$ head) $+$ rotate() method
(rotate() moves the first element to the end of the list)



Round-robin scheduler can be implemented using a circularly linked list C by repeatedly performing:

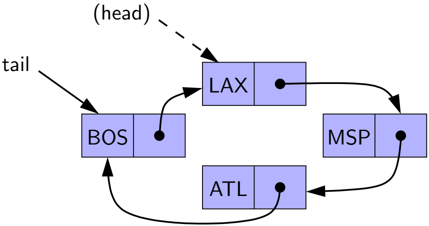
1. Give a time slice to process C.first()
2. C.rotate()

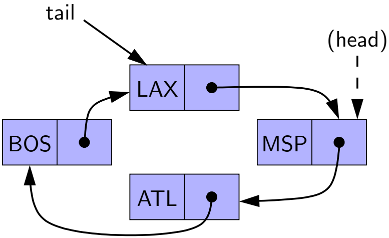
Additional optimization

- Head reference is no longer required.
  Head can be accessed as tail.getNext()
- Maintaining only the tail reference is simpler, time-, and space-efficient.
- This implementation is superior to singly linked list implementation.

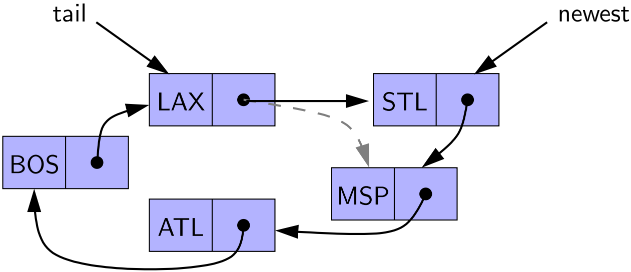
Rotate operation

- Simply advance the tail reference to its next node.





Insert element at head



CircularlyLinkedList class

```java
public class CircularlyLinkedList<E> {
  // nested node class identical to that of the SinglyLinkedList class
  private static class Node<E> {...}

  private Node<E> tail = null;       // we store tail (but not head)
  private int size = 0;              // number of nodes in the list
  public CircularlyLinkedList() { }  // constructs an initially empty list

  // access methods
  public int size() { return size; }
  public boolean isEmpty() { return size == 0; }
  public E first() {...}             // returns the first element
  public E last() {...}              // returns the last element

  // update methods
  public void rotate() {...}         // rotate the first element to the last
  public void addFirst(E e) {...}    // adds element e to the front
  public void addLast(E e) {...}     // adds element e to the end
  public E removeFirst() {...}       // removes and returns the first element
}
```

Access methods

```java
public E first() {                      // returns the first element
  if (isEmpty()) return null;
  return tail.getNext().getElement();   // the head is after the tail
}
```

```java
public E last() {                       // returns the last element
  if (isEmpty()) return null;
  return tail.getElement();
}
```

Update methods

```
public void rotate() {             // rotate the first element to the last
  if (tail != null)                // if empty, do nothing
    tail = tail.getNext();         // the old head becomes the new tail
}
```

```
public void addFirst(E e) {        // adds element e to the front of the list
  if (size == 0) {
    tail = new Node<>(e, null);
    tail.setNext(tail);            // link to itself circularly
  } else {
    Node<E> newest = new Node<>(e, tail.getNext());
    tail.setNext(newest);
  }
  size++;
}
```

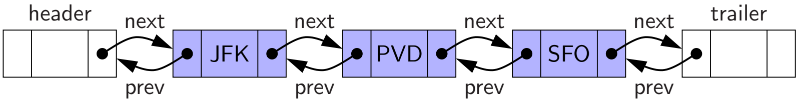
Update methods

```
public void addLast(E e) {           // adds element e to the end of the list
  addFirst(e);                       // insert new element at front of list
  tail = tail.getNext();             // now new element becomes the tail
}
```

```
public E removeFirst() {             // removes and returns the first element
  if (isEmpty()) return null;        // nothing to remove
  Node<E> head = tail.getNext();
  if (head == tail) tail = null;     // must be the only node left
  else tail.setNext(head.getNext()); // removes "head" from the list
  size--;
  return head.getElement();
}
```

Doubly Linked Lists

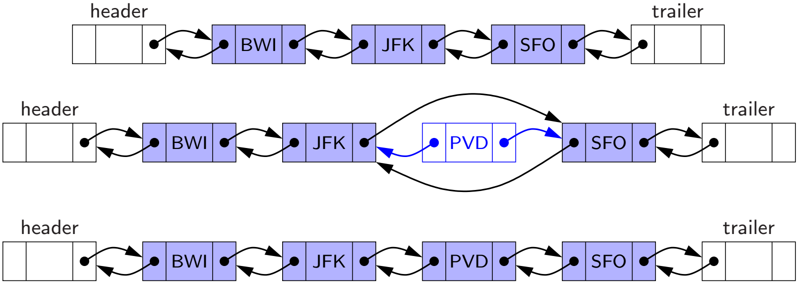
Doubly linked lists



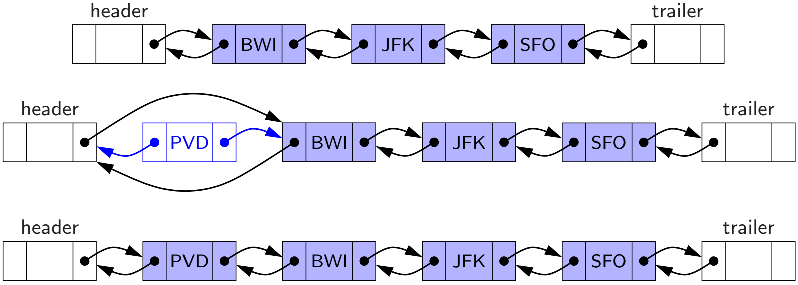
Advantages of using sentinels:

- Header and trailer nodes never change, only the nodes between them change
- Insertions and deletions can be handled in a unified manner

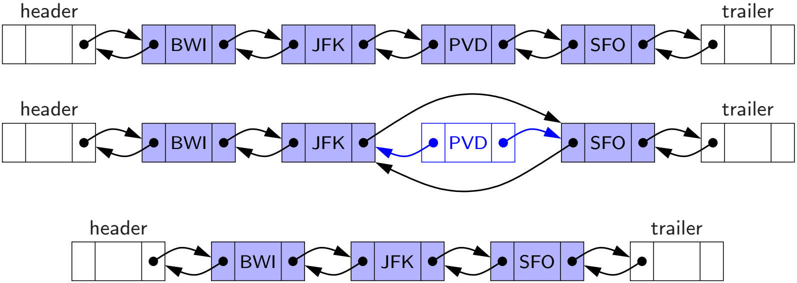
Inserting a node



Inserting at the front



Deleting a node



Methods for DoublyLinkedList class

Method	Functionality
`size()`	Returns the number of elements in the list.
`isEmpty()`	Returns true if the list is empty, and false otherwise.
`first()`	Returns the first element in the list.
`last()`	Returns the last element in the list.
`addFirst(e)`	Adds a new element to the front of the list.
`addLast(e)`	Adds a new element to the end of the list.
`removeFirst()`	Removes and returns the first element of the list.
`removeLast()`	Removes and returns the last element of the list.

DoublyLinkedList class

```
public class DoublyLinkedList<E> {
  // nested Node class
  private static class Node<E> {...}
  private Node<E> header;                    // header sentinel
  private Node<E> trailer;                   // trailer sentinel
  private int size = 0;

  // access methods
  public DoublyLinkedList() {...}
  public int size() {...}
  public boolean isEmpty() {...}
  public E first() {...}
  public E last() {...}

  // update methods
  public void addFirst(E e) {...}
  public void addLast(E e) {...}
  public E removeFirst() {...}
  public E removeLast() {...}
  // private update methods
  private void addBetween(E e, Node<E> predecessor, Node<E> successor) {...}
  private E remove(Node<E> node) {...}
}
```

Node class

```
//---------------- nested Node class ----------------
private static class Node<E> {
  private E element;            // reference to the element stored at this node
  private Node<E> prev;         // reference to the previous node in the list
  private Node<E> next;         // reference to the subsequent node in the list

  public Node(E e, Node<E> p, Node<E> n) {
    element = e; prev = p; next = n;
  }
  public E getElement() { return element; }
  public Node<E> getPrev() { return prev; }
  public Node<E> getNext() { return next; }
  public void setPrev(Node<E> p) { prev = p; }
  public void setNext(Node<E> n) { next = n; }
} //----------- end of nested Node class -----------
```

Access methods

```
public DoublyLinkedList() {
  header = new Node<>(null, null, null);      // create header
  trailer = new Node<>(null, header, null);   // trailer is preceded by header
  header.setNext(trailer);                    // header is followed by trailer
}
```

```
// public access methods
public int size() { return size; }
```

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

```
public E first() {
  if (isEmpty()) return null;
  return header.getNext().getElement();       // first element is beyond header
}
```

```
public E last() {
  if (isEmpty()) return null;
  return trailer.getPrev().getElement();      // last element is before trailer
}
```

Private update methods

```java
/* Adds an element to the linked list in between the given nodes. */
private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
  // create and link a new node
  Node<E> newest = new Node<>(e, predecessor, successor);
  predecessor.setNext(newest);
  successor.setPrev(newest);
  size++;
}
```

```java
/* Removes the given node from the list and returns its element. */
private E remove(Node<E> node) {
  Node<E> predecessor = node.getPrev();
  Node<E> successor = node.getNext();
  predecessor.setNext(successor);
  successor.setPrev(predecessor);
  size--;
  return node.getElement();
}
```

Update methods

```java
/* Adds an element to the front of the list. */
public void addFirst(E e) {
  addBetween(e, header, header.getNext());   // place just after the header
}
```

```java
/* Adds an element to the end of the list. */
public void addLast(E e) {
  addBetween(e, trailer.getPrev(), trailer); // place just before the trailer
}
```

```java
/* Removes and returns the first element of the list. */
public E removeFirst() {
  if (isEmpty()) return null;                // nothing to remove
  return remove(header.getNext());           // first element is beyond header
}
```

```java
/* Removes and returns the last element of the list. */
public E removeLast() {
  if (isEmpty()) return null;                // nothing to remove
  return remove(trailer.getPrev());          // last element is before trailer
}
```

Comparison table of linear data structures

Operation	Dyn. array	SLL/CLL	DLL
Insert first	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Insert last	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Insert between	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Insert at index	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Delete first	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Delete last	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Delete at index	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Access at index	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Modify at index	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Size	unlimited	unlimited	unlimited

List ADT

List ADT

Method	Functionality
`size()`	Returns the number of elements in the list.
`isEmpty()`	Returns a boolean indicating whether the list is empty.
`get(i)`	Returns the element of the list having index i; an error condition occurs if i is not in range [0,size() - 1].
`set(i, e)`	Replaces the element at index i with e, and returns the old element that was replaced; an error condition occurs if i is not in range [0,size() - 1].
`add(i, e)`	Inserts a new element e into the list so that it has index i, moving all subsequent elements one index later in the list; an error condition occurs if i is not in range [0,size()].
`remove(i)`	Removes and returns the element at index i, moving all subsequent elements one index earlier in the list; an error condition occurs if i is not in range [0,size() - 1]

Operations on a list

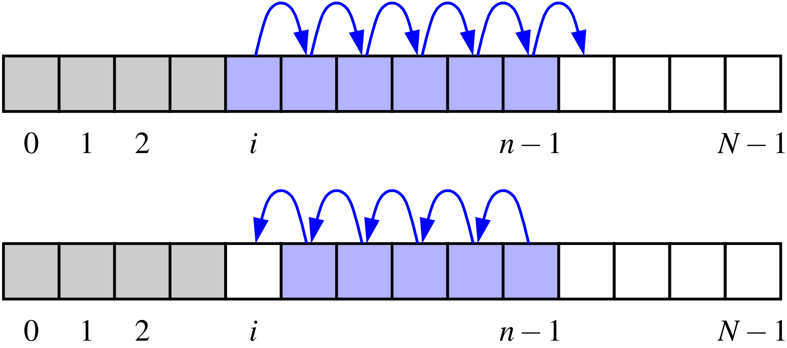
Method	Return value	List contents
add(0, A)	-	(A)
add(0, B)	-	(B, A)
get(1)	A	(B, A)
set(2, C)	error	(B, A)
add(2, C)	-	(B, A, C)
add(4, D)	error	(B, A, C)
remove(1)	A	(B, C)
add(1, D)	-	(B, D, C)
add(1, E)	-	(B, E, D, C)
get(4)	error	(B, E, D, C)
add(4, F)	-	(B, E, D, C, F)
set(2, G)	D	(B, E, G, C, F)
get(2)	G	(B, E, G, C, F)

Simplified java.util.List interface

```java
/* A simplified version of the java.util.List interface. */
public interface List<E> {
  int size();
  boolean isEmpty();

  /* Returns (but does not remove) the element at index i. */
  E get(int i) throws IndexOutOfBoundsException;

  /* Replaces element at index i with e, and returns replaced element. */
  E set(int i, E e) throws IndexOutOfBoundsException;

  /* Inserts e to be at index i, shifting subsequent elements later. */
  void add(int i, E e) throws IndexOutOfBoundsException;

  /* Removes the element at index i, shifting subsequent elements earlier. */
  E remove(int i) throws IndexOutOfBoundsException;
}
```

Array Lists

- Implement the List ADT using an array.
- Get/set methods are fast, but add/remove methods are slow.



Simple ArrayList implementation

```java
public class ArrayList<E> implements List<E> {
  public static final int CAPACITY=16;     // default array capacity
  private E[] data;                        // generic array used for storage
  private int size = 0;                    // current number of elements

  public ArrayList() { this(CAPACITY); }   // constructs list with default cap.
  public ArrayList(int capacity) { data = (E[]) new Object[capacity]; }

  public int size() { return size; }
  public boolean isEmpty() { return size == 0; }
  public E get(int i) throws IndexOutOfBoundsException {...}
  public E set(int i, E e) throws IndexOutOfBoundsException {...}
  public void add(int i, E e) throws IndexOutOfBoundsException {...}
  public E remove(int i) throws IndexOutOfBoundsException {...}

  // utility methods
  /** Checks whether the given index is in the range [0, n-1]. */
  protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {...}
  /** Resizes internal array to have given capacity >= size. */
  protected void resize(int capacity) {...}
}
```

Access methods

```
/* Returns (but does not remove) the element at index i. */
public E get(int i) throws IndexOutOfBoundsException {
  checkIndex(i, size);
  return data[i];
}
```

```
/* Replaces the element at the specified index, and
 * returns the element previously stored. */
public E set(int i, E e) throws IndexOutOfBoundsException {
  checkIndex(i, size);
  E temp = data[i];
  data[i] = e;
  return temp;
}
```

Update methods

```
/* Inserts the given element at the specified index of the list, shifting all
 * subsequent elements in the list one position further to make room. */
public void add(int i, E e) throws IndexOutOfBoundsException {
  checkIndex(i, size + 1);
  if (size == data.length)               // not enough capacity
    resize(2 * data.length);             // so double the current capacity
  for (int k=size-1; k >= i; k--)        // start by shifting rightmost
    data[k+1] = data[k];
  data[i] = e;                           // ready to place the new element
  size++;
}
```

```
/* Removes and returns the element at the given index, shifting all subsequent
 * elements in the list one position closer to the front. */
public E remove(int i) throws IndexOutOfBoundsException {
  checkIndex(i, size);
  E temp = data[i];
  for (int k=i; k < size-1; k++)         // shift elements to fill hole
    data[k] = data[k+1];
  data[size-1] = null;                   // help garbage collection
  size--;
  return temp;
}
```

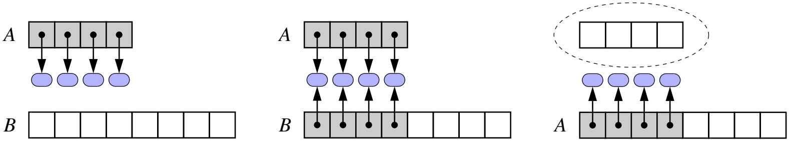
Utility methods

```java
/* Checks whether the given index is in the range [0, n-1]. */
protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {
  if (i < 0 || i >= n)
    throw new IndexOutOfBoundsException("Illegal index: " + i);
}
```

```java
/* Resizes internal array to have given capacity >= size. */
protected void resize(int capacity) {
  E[] temp = (E[]) new Object[capacity];     // safe cast
  for (int k=0; k < size; k++)
    temp[k] = data[k];
  data = temp;                               // start using the new array
}
```

Dynamic array

- Adding elements leads to the overflow problem
- The overflow problem can be handled by growing the array



GROW-ARRAY(A, n)

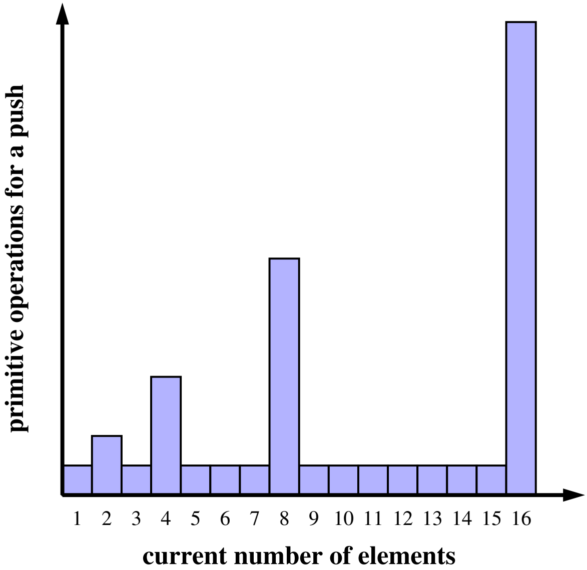
1. Allocate a new array B with larger capacity.
2. Set $B[k] = A[k]$, for $k \leftarrow 0, \ldots, n-1$, where n denotes current number of items.
3. Set $A \leftarrow B$, that is, we henceforth use the new array to support the list.
4. Leave the old array to be garbage collected.

Dynamic array using array doubling

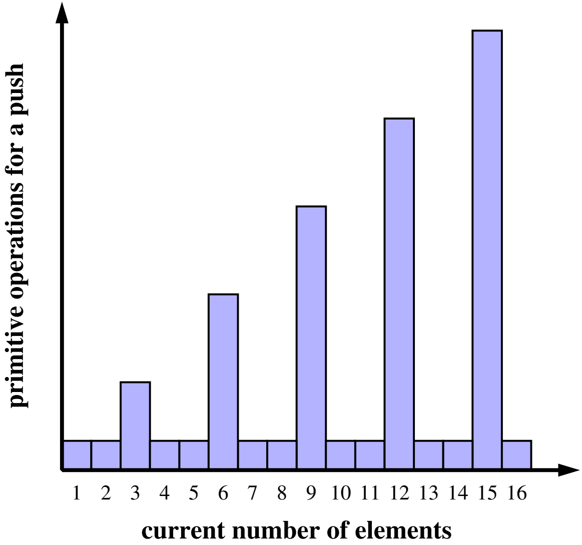
```
/* Resizes internal array to have given capacity >= size. */
protected void resize(int capacity) {
  E[] temp = (E[]) new Object[capacity];     // safe cast
  for (int k=0; k < size; k++)
    temp[k] = data[k];
  data = temp;                               // start using the new array
}
```

```
/* Inserts the given element at the specified index of the list, shifting all
 * subsequent elements in the list one position further to make room. */
public void add(int i, E e) throws IndexOutOfBoundsException {
  checkIndex(i, size + 1);
  if (size == data.length)               // not enough capacity
    resize(2 * data.length);             // so double the current capacity
  // rest of the method
}
```

Functions for growing/resizing arrays



Functions for growing/resizing arrays



Amortized analysis of dynamic arrays

- Amortized analysis.

  Show that performing a sequence of operations is quite efficient
- Core idea.

  Instead of considering worst-case time taken per operation, consider the average time taken per operation.

Amortized analysis of dynamic arrays

- Use geometric progressions
  $\langle a, ar, ar^2, \ldots \rangle$, such that $r \in \mathbb{R}$ and $r > 1$
  Total time to perform n add operations is $\Theta\left(n\right)$
  The value r chosen depends on the trade-off between runtime efficiency and memory usage
- Do not use arithmetic progressions
  $\langle a, a + d, a + 2d, \ldots \rangle$, such that $d \in \mathbb{N}$
  Total time to perform n add operations is $\Theta\left(n^2\right)$

Shrinking the dynamic array

- What if you repeatedly remove elements from an arbitrarily large array?
- What if there is an oscillation between growing and shrinking the underlying array?
- The array capacity is halved when the number of elements falls below $1/4$th of the capacity

String vs. StringBuilder class

```java
public String repeat1(char c, int n) {
  String answer = "";
  for (int j=0; j < n; j++)
    answer += c;
  return answer;
}
```

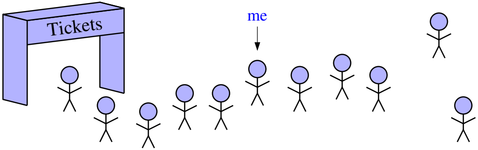
- Static array
- Resize every time
- Time for n adds is $\Theta\left(n^2\right)$

```java
public String repeat2(char c, int n) {
  StringBuilder sb = new StringBuilder();
  for (int j=0; j < n; j++)
    sb.append(c);
  return sb.toString();
}
```

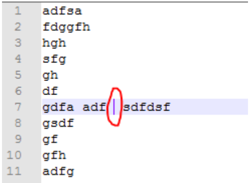
- Dynamic array
- Resize a few times
- Time for n adds is $\Theta\left(n\right)$

Positional Lists

Location/position in a sequence



- Position in a queue = node reference



```
adfsa
fdggfh
hgh
sfg
gh
df
gdfa adf|sdfdsf
gsdf
gf
gfh
adfg
```

- Cursor in a text editor = node reference

Positional list ADT

Method	Functionality
getElement()	Returns the element stored at this position.

- The position of an element does not change even its index changes due to insertions/deletions in the list

Accessor methods in a positional list

Method	Functionality
`first()`	Returns the position of the first element of L (or null if empty).
`last()`	Returns the position of the last element of L (or null if empty).
`before(p)`	Returns the position of L immediately before position p (or null if p is the first position).
`after(p)`	Returns the position of L immediately after position p (or null if p is the last position).
`isEmpty()`	Returns true if list L does not contain any elements.
`size()`	Returns the number of elements in list L.

A traversal of a positional list

```java
Position<String> cursor = guests.first();
while (cursor != null) {
  System.out.println(cursor.getElement());
  cursor = guests.after(cursor);     // advance to the next position (if any)
}
```

Update methods in a positional list

Method	Functionality
addFirst(e)	Inserts a new element e at the front of the list, returning the position of the new element.
addLast(e)	Inserts a new element e at the back of the list, returning the position of the new element.
addBefore(p,e)	Inserts a new element e in the list, just before position p, returning the position of the new element.
addAfter(p,e)	Inserts a new element e in the list, just after position p, returning the position of the new element.
set(p,e)	Replaces the element at position p with element e, returning the element formerly at position p.
remove(p)	Removes and returns the element at position p in the list, invalidating the position.

Operations on a positional list

Method	Return value	List contents
addLast(8)	p	(8p)
first()	p	(8p)
addAfter(p, 5)	q	(8p, 5q)
before(q)	p	(8p, 5q)
addBefore(q, 3)	r	(8p, 3r, 5q)
getElement()	3	(8p, 3r, 5q)
after(p)	r	(8p, 3r, 5q)
before(p)	null	(8p, 3r, 5q)
addFirst(9)	s	(9s, 8p, 3r, 5q)
remove(last())	5	(9s, 8p, 3r)
set(p, 7)	8	(9s, 7p, 3r)
remove(q)	error	(9s, 7p, 3r)

Java interface for positional list ADT

```java
public interface Position<E> {
  E getElement() throws IllegalStateException;
}
```

```java
public interface PositionalList<E> {
  int size();
  boolean isEmpty();
  Position<E> first();
  Position<E> last();
  Position<E> before(Position<E> p) throws IllegalArgumentException;
  Position<E> after(Position<E> p) throws IllegalArgumentException;
  Position<E> addFirst(E e);
  Position<E> addLast(E e);
  Position<E> addBefore(Position<E> p, E e) throws IllegalArgumentException;
  Position<E> addAfter(Position<E> p, E e) throws IllegalArgumentException;
  E set(Position<E> p, E e) throws IllegalArgumentException;
  E remove(Position<E> p) throws IllegalArgumentException;
}
```

Positional list using DLL

```
public class LinkedPositionalList<E> implements PositionalList<E> {
  private static class Node<E> implements Position<E> {...}
  private Node<E> header;             // header sentinel
  private Node<E> trailer;            // trailer sentinel
  private int size = 0;

  public LinkedPositionalList() {...}
  private Node<E> validate(Position<E> p) throws IllegalArgExcep {...}
  private Position<E> position(Node<E> node) {...}
  public int size() {...}
  public boolean isEmpty() {...}
  public Position<E> first() {...}
  public Position<E> last() {...}
  public Position<E> before(Position<E> p) throws IllegalArgExcep {...}
  public Position<E> after(Position<E> p) throws IllegalArgExcep {...}
  private Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {...}
  public Position<E> addFirst(E e) {...}
  public Position<E> addLast(E e) {...}
  public Position<E> addBefore(Position<E> p, E e) throws IllegalArgExcep {...}
  public Position<E> addAfter(Position<E> p, E e) throws IllegalArgExcep {...}
  public E set(Position<E> p, E e) throws IllegalArgExcep {...}
  public E remove(Position<E> p) throws IllegalArgExcep {...}
}
```

Positional list using DLL

```java
//---------------- nested Node class ----------------
private static class Node<E> implements Position<E> {
  private E element;        // reference to the element stored at this node
  private Node<E> prev;     // reference to the previous node in the list
  private Node<E> next;     // reference to the subsequent node in the list
  public Node(E e, Node<E> p, Node<E> n)
  { element = e; prev = p; next = n; }

  // public accessor methods
  public E getElement() throws IllegalStateException {
    if (next == null)       // convention for defunct node
      throw new IllegalStateException("Position no longer valid");
    return element;
  }
  public Node<E> getPrev() { return prev; }
  public Node<E> getNext() { return next; }
  public void setElement(E e) { element = e; }
  public void setPrev(Node<E> p) { prev = p; }
  public void setNext(Node<E> n) { next = n; }
} //----------- end of nested Node class -----------
```

Constructor and private utilities

```
public LinkedPositionalList() {
  header = new Node<>(null, null, null);      // create header
  trailer = new Node<>(null, header, null);   // trailer is preceded by header
  header.setNext(trailer);                    // header is followed by trailer
}
```

```
private Node<E> validate(Position<E> p) throws IllegalArgumentException {
  if (!(p instanceof Node)) throw new IllegalArgumentException("Invalid p");
  Node<E> node = (Node<E>) p;     // safe cast
  if (node.getNext() == null)     // convention for defunct node
    throw new IllegalArgumentException("p is no longer in the list");
  return node;
}
```

```
private Position<E> position(Node<E> node) {
  if (node == header || node == trailer)
    return null;   // do not expose user to the sentinels
  return node;
}
```

Accessor methods

```java
public Position<E> first() { return position(header.getNext()); }
```

```java
public Position<E> last() { return position(trailer.getPrev()); }
```

```java
public Position<E> before(Position<E> p) throws IllegalArgumentException {
  Node<E> node = validate(p);
  return position(node.getPrev());
}
```

```java
public Position<E> after(Position<E> p) throws IllegalArgumentException {
  Node<E> node = validate(p);
  return position(node.getNext());
}
```

Update methods

```java
private Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
  Node<E> newest = new Node<>(e, pred, succ);  // create and link a new node
  pred.setNext(newest); succ.setPrev(newest); size++;
  return newest;
}
```

```java
public Position<E> addFirst(E e)
{ return addBetween(e, header, header.getNext()); }
```

```java
public Position<E> addLast(E e)
{ return addBetween(e, trailer.getPrev(), trailer); }
```

```java
public Position<E> addBefore(Position<E> p, E e)
                                throws IllegalArgumentException {
  Node<E> node = validate(p);
  return addBetween(e, node.getPrev(), node);
}
```

```java
public Position<E> addAfter(Position<E> p, E e)
                            throws IllegalArgumentException {
  Node<E> node = validate(p);
  return addBetween(e, node, node.getNext());
}
```

Update methods

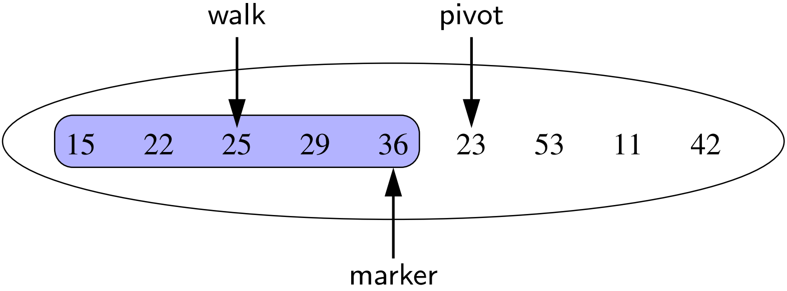
```
public E set(Position<E> p, E e) throws IllegalArgumentException {
  Node<E> node = validate(p);
  E answer = node.getElement();
  node.setElement(e);
  return answer;
}
```

```
public E remove(Position<E> p) throws IllegalArgumentException {
  Node<E> node = validate(p);
  Node<E> predecessor = node.getPrev();
  Node<E> successor = node.getNext();
  predecessor.setNext(successor);
  successor.setPrev(predecessor);
  size--;
  E answer = node.getElement();
  node.setElement(null);          // help with garbage collection
  node.setNext(null);             // and convention for defunct node
  node.setPrev(null);
  return answer;
}
```

Performance of a linked positional list

Method	Running time
size()	$\mathcal{O}(1)$
isEmpty()	$\mathcal{O}(1)$
first(), last()	$\mathcal{O}(1)$
before(p), after(p)	$\mathcal{O}(1)$
addFirst(e), addLast(e)	$\mathcal{O}(1)$
addBefore(p, e), addAfter(p, e)	$\mathcal{O}(1)$
set(p, e)	$\mathcal{O}(1)$
remove(p)	$\mathcal{O}(1)$

Sorting a positional list



Sorting a positional list

```java
/** Insertion-sort of a positional list of integers into nondecreasing order */
public static void insertionSort(PositionalList<Integer> list) {
  Position<Integer> marker = list.first(); // last position known to be sorted
  while (marker != list.last()) {
    Position<Integer> pivot = list.after(marker);
    int value = pivot.getElement(); // number to be placed
    if (value > marker.getElement()) // pivot is already sorted
      marker = pivot;
    else { // must relocate pivot
      Position<Integer> walk = marker; // find leftmost item greater than value
      while (walk != list.first() && list.before(walk).getElement() > value)
        walk = list.before(walk);
      list.remove(pivot); // remove pivot entry and
      list.addBefore(walk, value); // reinsert value in front of walk
    }
  }
}
```