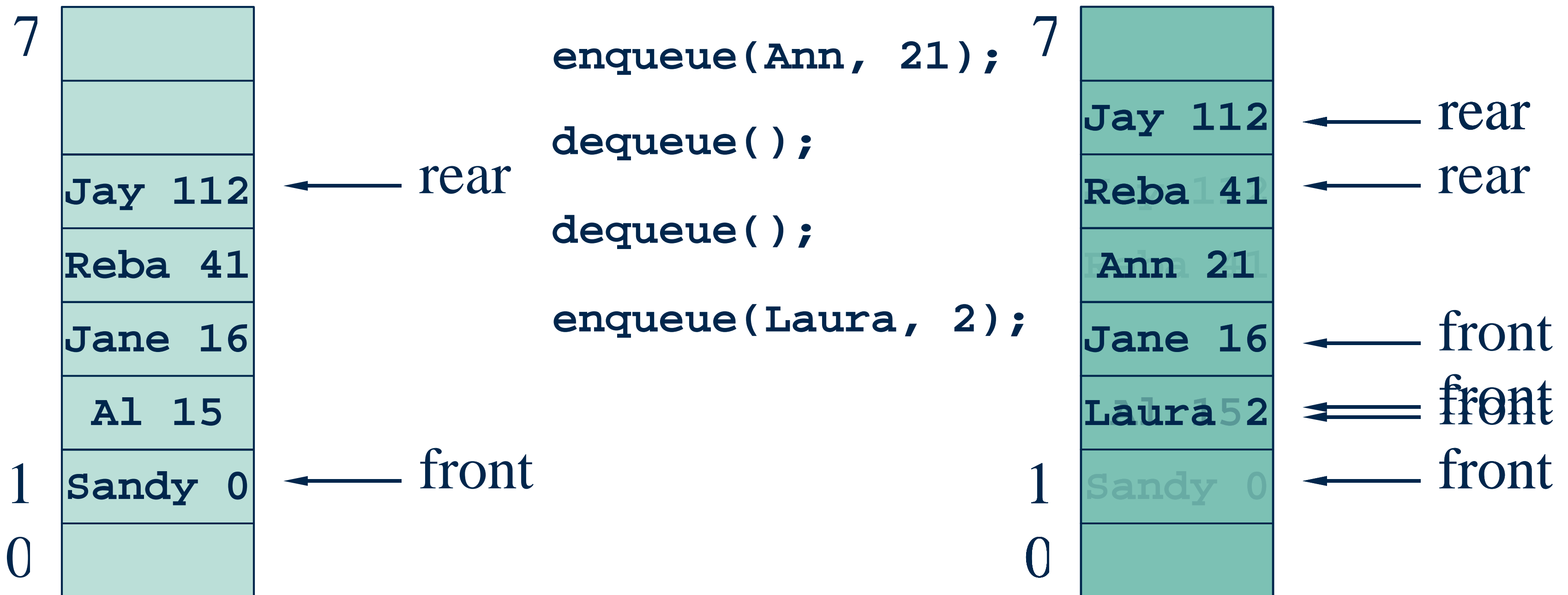
Priority Queue

- A specialized queue where the order of the queue is kept according to priorities.
- Insertion of an object into the queue (enqueue) no longer always happens to the end of the queue.
- Object is inserted into the appropriate position in the queue based on its priority.

Example: Priority Queues

- Keep the queue in sorted order



Other implementations of PQs

- Leave the queue unsorted, but always dequeue the item with the smallest priority
- Use multiple queues of different priority settings
  - Since each queue has all objects of the same priority, we go back to a normal queue
  - When dequeueing, always take off from the lowest non-empty priority queue

Asymptotic Efficiency

- Stack
  - Push – O(1) – doubling aside
  - Pop – O(1)
- Queue
  - Enqueue – O(1) – doubling aside
  - Dequeue – O(1)
- Priority Queue
  - Enqueue – O(n) – doubling aside
  - Dequeue -- O(1)

Stack Applications

- Stacks are often used to remember where we were (i.e. back tracking), or to revert back to a previous state.
- Some common stack applications:
  - language parsing
  - undo features, scratch pads, path finding
  - function/methods calls

Reversing a Word

- Idea: push each character of the word on to the stack as one reads them from left to right. When done, pop the characters off the stack.

Word Reverse

```
  public void push(char c) {
      theArray[++tos] = c;
  }
  public char pop() {
      if (isEmpty() != true)
        return theArray[tos--];
  }
  public char top() {
      if (isEmpty() != true)
        return theArray[tos];
  }
}
```

Word Reverse

```
class Reverser {
  private String input;
  private String output = “”;
  public Reverser(String in){input=in;}
  public String reverse() {
      int s = input.length;
      MyStack s = new MyStack(s);
      for (int i=0;i<s; i++)
        s.push(input.charAt(i));
      while (s.isEmpty() != true)
        output += s.pop();
      return output;
   }
}
```

Delimiter Matching

- Parsing the parentheses in a programming language such as Java
- Parsing the parentheses in a math expression
  - `c[d]               // correct`
  - `a{b[[c]d]}         // correct`
  - `a{b[c}d}           // incorrect`
  - `a{b(c)             // incorrect`

Delimiter Matching

```
class CharStack {
  private char[] theArray;
  private int size;
  private int tos;  //top of stack -1 if empty

  public MyStack(int size) {
    theArray = new char[size];
    tos = -1;
  }
  public boolean isEmpty() {
     return (tos == -1);
  }
  ...
```

What are the rules of of matching parentheses?

- Every open paren must be matched by exactly one closing one.
- The last occuring paren must be the first which is matched (closed), that is, all closing must be done in the reverse order in which it was opened.

Solution with a Stack

- Start reading the text from left to right
- When we see an open paren, push it onto the stack.
- When we see a closing paren, look at the current top of stack. If the current tos paren matches the closing paren, then pop the tos and continue. Otherwise report non-matching.
- Do nothing for all other characters.
- If at the end of input, stack is empty, then matching is successful.

a{b(cc[d])e}f

char read	stack
a	-
{	{
b	{
(	{(
c	{(
c	{(
[	{([
d	{([
]	{(
)	{
e	{
}	-
f	-

Delimiter Matching

```
public class ParenMatch {
    public static void main(String[] args) {
       char[] open = {'<', '[', '{', '('};
       char[] clos = {'>', ']', '}', ')'};
       CharStack cs;
       try {
              for (int i=0; i<args.length; i++) {
                     cs = new CharStack();
                     boolean ok=true;
                     for (int j=0; j<args[i].length(); j++) {
                            char c = args[i].charAt(j);
                            boolean got=false;
                            for (int z=0; z<open.length; z++)
                                if (c==open[z]) got=true;
                            if (got) {
                                cs.push(c);
                            }
                            else {
                                   for (int z=0; z<clos.length; z++)
                                       if (c==clos[z]) got=true;
                                   if (got) {
                                          if (!(cs.pop()==c))
                                              ok=false;
                                       }   }   }
                     System.out.println(args[i] + " " + ok);  }  }
       catch (Exception e)  {
              e.printStackTrace(); }   }   }
```

Classing the Registrar