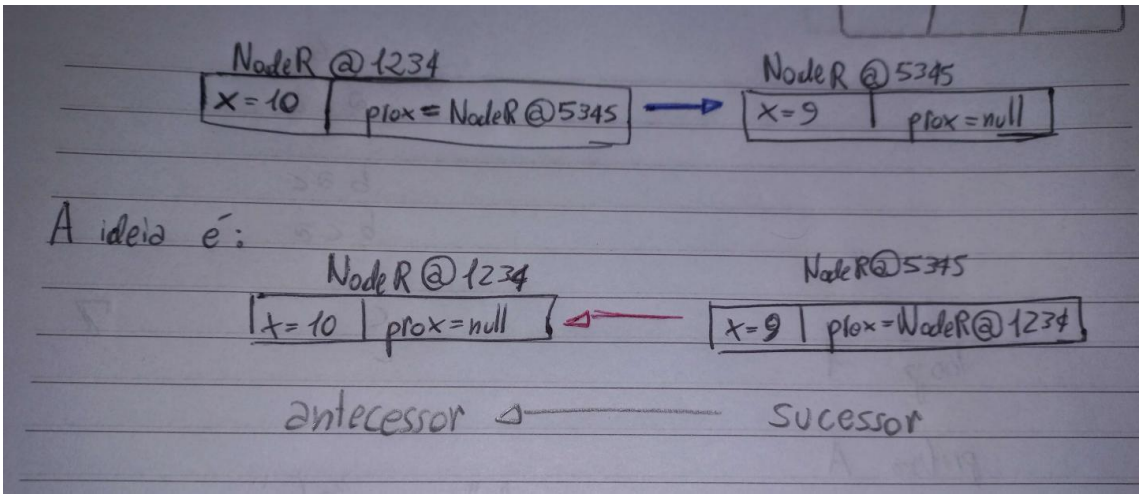
Lista Encadeada com Recursividade:

Método Inverte

Marco Bockoski – 6EC - UNITAU

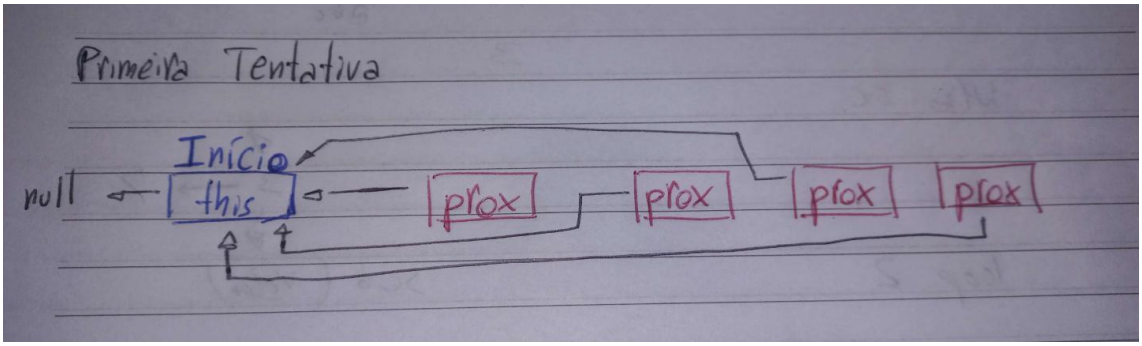
Ideia resolutiva do método:

- Inversão da referência prox de cada elemento: trocar a referência do sucessor para o antecessor.
- Atualizar o inicio/head do nó.



PRIMEIRA TENTATIVA

```
public void inverte(){
        if(prox.getProx() != null){
                prox.inverte();
        }
        prox.setProx(this)
}
```



O problema dessa resolução se divide em dois:

- Não se modifica o nó que corresponde ao início/head, portanto, ao invocar métodos como show(), a lista irá corresponder apenas ao último nó, que tem sua referência voltada ao null.
- A ideia de adotar a notação “this” seria válida se ela correspondesse ao objeto que executa o método, porém, essa notação corresponde unicamente ao início/head, portanto, o que acontece é que todos os nós passam a apontar para o início/head, como se fosse uma “árvore invertida”.

SEGUNDA TENTIVA

Resultado direto do fracasso da primeira tentativa, o método inverte() é constituído de outro método recursivo denominado getNodeAt() que serve tanto para corrigir o uso da notação “this” quanto a devida inversão da lista sem haver erros, como eventualmente será mostrado uma situação propícia para Stack Overflow durante a execução do método size().

Método getNodeAt()

- Este Método tem como valor de retorno um NodeR
- Como argumento, esse método recebe um valor corresponde à posição do nó que se deseja.
- Portanto sua assinatura fica: public NodeR getNodeAt(int pos).

As soluções triviais para esse método recursivo são duas:

- Se deseja o NodeR da posição 1, return this (correspondete ao head/início)
- Se deseja o NodeR da posição 2, return prox (o elemento logo após o head/início.

Como as outras posições podem ser obtidas? A notação “this” pode funcionar apenas para o head/início, porém a variável prox do NodeR não, e justamente é por ela que a recursão é executada.

O elemento da posição pretendida corresponde prox, enquanto como argumento, “pos” pode receber valores maiores que 2.

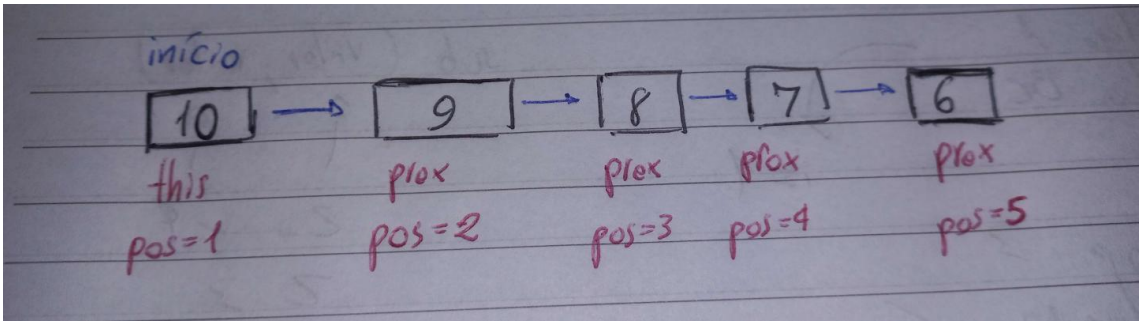
Conclusão: Deve haver uma chamada recursiva quando “pos > 2”, de modo que “pos” deve decrementar quantidade suficiente para atender a exigência de “pos == 2” que retorna prox.

Tal explicação está presente no código na seguinte sintaxe:

if(pos == 1) return this;

if(pos == 2) return prox;

else prox.getNodeAt(pos-1);



Existem alguns detalhes periféricos que complementam o método inteiramente:

- Se o argumento for um valor negativo ou um valor maior que o size(), de modo que não pertence à lista?
- Se a lista for vazia, o que deve acontecer?
- Como é implementado o return desse método?

Utilizando o método size(), é aferido o tamanho da lista e sabendo que o primeiro elemento corresponde ao 1, o argumento do método deve ser um inteiro maior que 0 e menor ou igual a size(). Portanto, todo código correspondente ao funcionamento regular do método é envolvido por:

If(pos <= limite && pos > 0)

Sendo limite: int limite = size();

No caso em que o argumento não está abrangido no if, um else deve ser executado enviando um exceção nomeada InvalidIndexException exclusiva da ListaRecursividade para evitar ser executado o código com argumento fora dos limites. Toda chamada do getNodeAt() é envolvido por try, sua falha resulta em um catch que inviabiliza a execução do método e informa ao usuário que não é possível ser feita. Sendo o valor de retorno indiscutivelmente um NodeR, quando tal situação ocorre, o método deve retornar null.

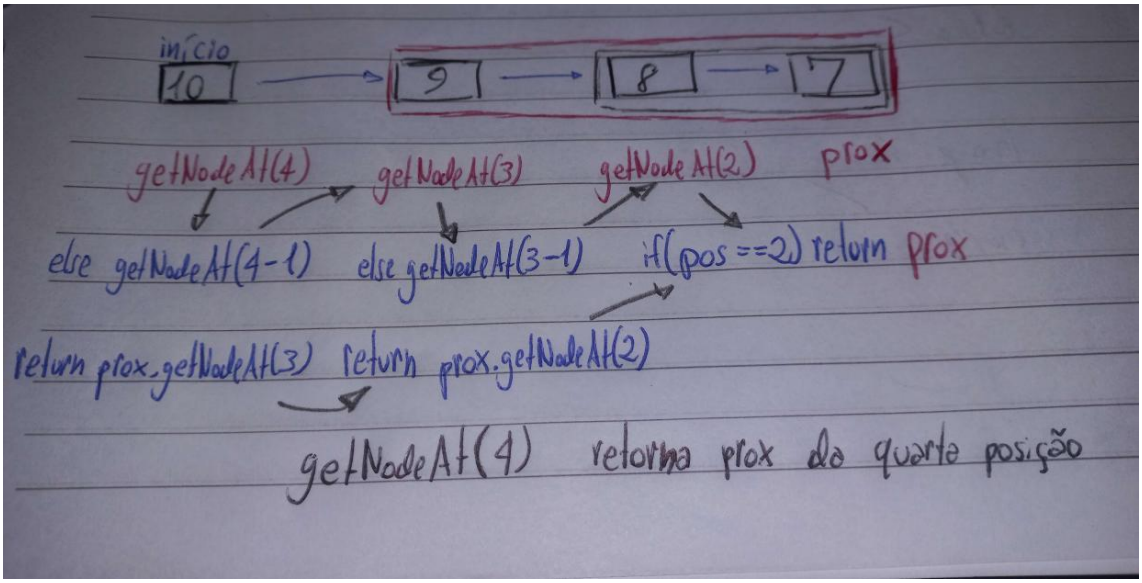
Assim, quando é feita com um argumento adequado, o método retorna prox.getNodeAt(pos-1). É mais fácil entender o porquê desse retorno com um exemplo, assim, considere o código completo:

```
public NodeR getNodeAt(int pos) throws InvalidIndexException{
        int limite = this.size();
        if((pos <= limite)&&(pos > 0)){
            if(pos == 1) return this;
            if(pos == 2) return prox;
            else prox.getNodeAt(pos-1);
        }else{
            throw new InvalidIndexException("\n\nInvalidIndexException: Impossivel realizar operacao, resultado vai dar null (index fora dos limites da lista)\n\n");
        }
        if((pos <= limite)&&(pos > 0)) return prox.getNodeAt(pos-1);
        return null; }
```

Em verde, está a assinatura do código, size() está destacada em azul. Destacado de roxo corresponde ao procedimento padrão do código, enquanto vermelho em caso de lista vazia ou argumento inválido.

- Considerando um argumento válido como pos = 4.
- O código será executado pela parte roxa.
- O método executado pelo NodeR head/início vai atender aos requisitos do else (pos é diferente de 1 e 2).
- Assim, o NodeR prox vai executar o mesmo código (recursivamente) com argumento pos = 3, que ainda é diferente de 1 e 2, caí na condição do else novamente.

- Finalmente, um NodeR prox vai executar também o mesmo código (recursivamente) com parâmetro pos = 2, que atende ao requisito do segundo if e retorna seu prox (o quarto nó).
- Retornando ao nó NodeR de argumento 3, seu valor de retorno corresponde à prox.getNodeAt(pos-1), portanto, seu valor de retorno é o Nó que foi o valor de retorno do nó de argumento pos = 2.
- O mesmo ocorre com o nó NodeR de argumento 4, assim, obtém-se o nó da quarta posição.



Observação recursiva: O getNodeAt recursivamente transforma uma lista de size()-1 gradativamente, até chegar a uma lista de 2 nós, para retorna o prox, que é o valor desejado.

Método inverte()

Agora já resolvida a pendência em relação a notação “this”, só resta trocar as partes que utilizam “this” para um getNodeAt, não? Não.

Utilizando exatamente o mesmo código que a primeira tentativa executa, um erro à princípio inesperado acontece, no entanto, é totalmente justificável a sua ocorrência:

```
public void inverte(int pos){
        if(pos != 1){
                getNodeAt(pos) .setProx(getNodeAt(pos-1));
                inverte(pos-1);
        }else{
                getNodeAt(1).setProx(null);
        }
}
```

Observando as linhas destacadas, as avermelhadas correspondem à inversão de referência dos objetos, enquanto a azulada corresponde a recursividade do método.

Percebe-se que a lista é invertida de trás pra frente, começando com “pos” em seu maior valor até chegar a pos = 1, no qual, esse nó terá a referência de null, indicando ser o novo fim da lista.

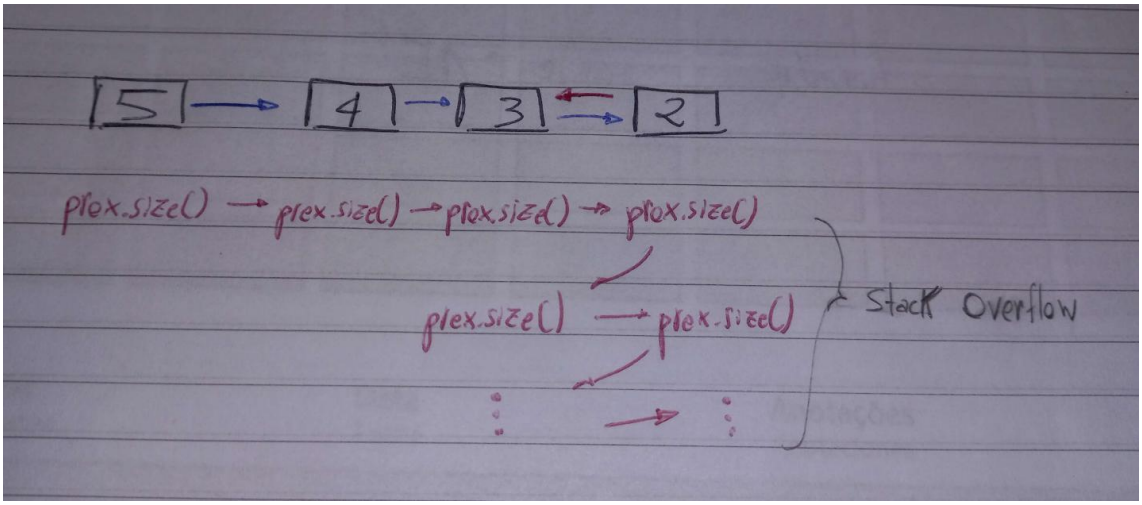
No entanto, apenas o último nó (de pos valor máximo) conseguirá ser invertido, e o compilador acusará Stack Overflow na execução de um método size().

O método size() é interno ao getNodeAt(), portanto é válido repensar o impacto da inversão e em qual linha exatamente implica, revendo o código de getNodeAt():

```
public NodeR getNodeAt(int pos) throws InvalidIndexException{
        int limite = this.size();
        if((pos <= limite)&&(pos > 0)){
            if(pos == 1) return this;
```

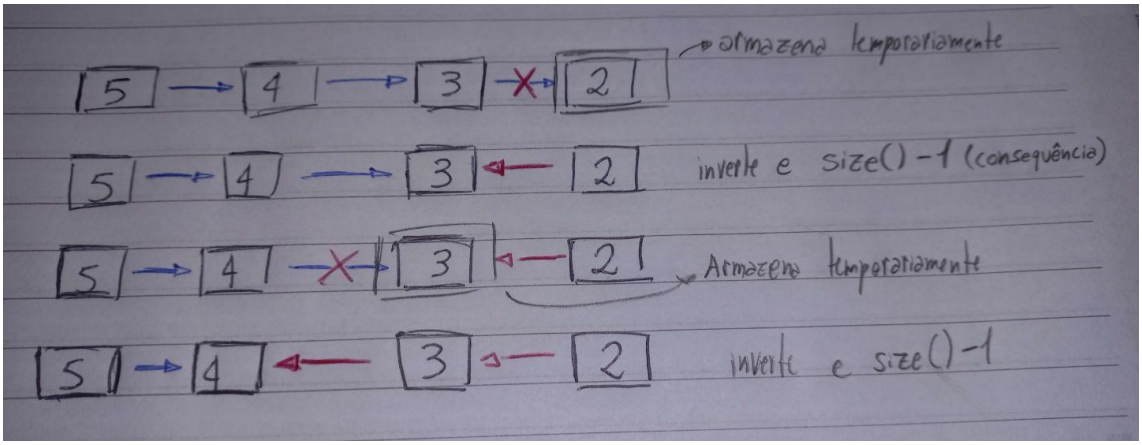
```
            if(pos == 2) return prox;
            else prox.getNodeAt(pos-1);
        }else{
            throw new InvalidIndexException("\n\nInvalidIndexException:
Impossivel realizar operacao, resultado vai dar null (index fora dos limites da
lista)\n\n");
        }
        if((pos <= limite)&&(pos > 0)) return prox.getNodeAt(pos-1);
        return null;
}
```

Precisamente, toda chamada do getNodeAt(), o método envolve o cálculo de size() para garantir o respeito dos limites da lista. Sabe-se que esse método é afetado na inversão dos nós (algumas referências são criadas ou retiradas), no entanto a inversão do último elemento faz com que a lista adquira um loop: cada nó NodeR faz referência um ao outro, de modo que prox.size() fique em loop, Stack Overflow é acusado decorrente disso, size() adquire um valor infinito portanto não determinável e de valor muito grande (tendência ao infinito).



A resolução desse problema é evitar o loop:

- Cortar a referência direta para criar a referência inversa:
  - Uma variável temporária NodeR deve ser criada para não perder o nó que teve sua referência cortada.
  - Após a variável temporária ter o valor de NodeR, a referência é cortada com o comando setProx(null)
  - A variável temporária tem sua referência definida por setProx(getNodeAt(pos-1))
- Após a inversão de uma ligação ser feita, recursivamente o método chama a si mesmo com o decremento de seu argumento pos.
- A lista fica de tamanho menor, mas isso também não é problema, pois o getNodeAt() está recebendo parâmetros decrescentes, como é observável na linha destacada de azul em seu código.



Assim, seu código com a variável temporária fica assim:

```
public void inverte(int pos){
        if(pos != 1){
            try{
                NodeR lastSalvado = getNodeAt(pos);
                getNodeAt(pos-1).setProx(null);
                lastSalvado.setProx(getNodeAt(pos-1));
            }catch(InvalidIndexException iie){
                System.out.println(iie.getMessage());
```

```
            }
                inverte(pos-1); //Parte da recursão

            }else{
                try{
                    getNodeAt(1).setProx(null);
                }catch(InvalidIndexException iie){
                    System.out.println(iie.getMessage());
                }
            }
        }
```

- Em verde corresponde a sua assinatura
- Em amarelo, corresponde a relação do NodeR ser o início/head ou não. (Se for, sua referência deve ser setada para null)
- Em azul, corresponde ao funcionamento regular da lógica, com o lastSalvado sendo o nó temporário. Ocorre corte da referência direta, criação da referência inversa e recursão em inverte(pos-1). Tudo envolto por try, evitando uma circunstância na qual getNodeAt() recebe um parâmetro disfuncional.
- Em vermelho, consequências diretas da má execução do que ocorre no que está envolto em try, caso getNodeAt() receba um parâmetro disfuncional ao usuário é informado que tal comando não é executável.

Métodos em ListaR

```
public NodeR getNodeAt(int valor){
        if(inicio!=null){
            try{
                return inicio.getNodeAt(valor);
            }catch(InvalidIndexException iie){
                System.out.println(iie.getMessage());
            }
        }
        return null;
    }
```

- Semelhante ao que ocorre no método inverte() em NodeR, o método tem assinatura, tem um critério para notar se a lista não está vazia e deve aferir se o parâmetro do valor é um valor válido, procede da mesma maneira para informar ao usuário.

```
public void inverte(){
        if((inicio!=null)&&(size()!=1)){
                int tamanho = inicio.size();
                NodeR temp = getNodeAt(tamanho);
                inicio.inverte(tamanho);
                inicio = temp;
         }else if(size()==1){
                System.out.println("Nao eh possivel inverter lista de apenas um no");
          }
      }
```

- As partes em verde e amarela são auto-explicativas.
- A parte em roxa faz o processo de inversão da lista, fornecendo size() como parâmetro.
- A parte em azul corresponde ao armazenamento temporário do último valor da lista, para se tornar eventualmente o novo início da lista quando a inversão estiver completa