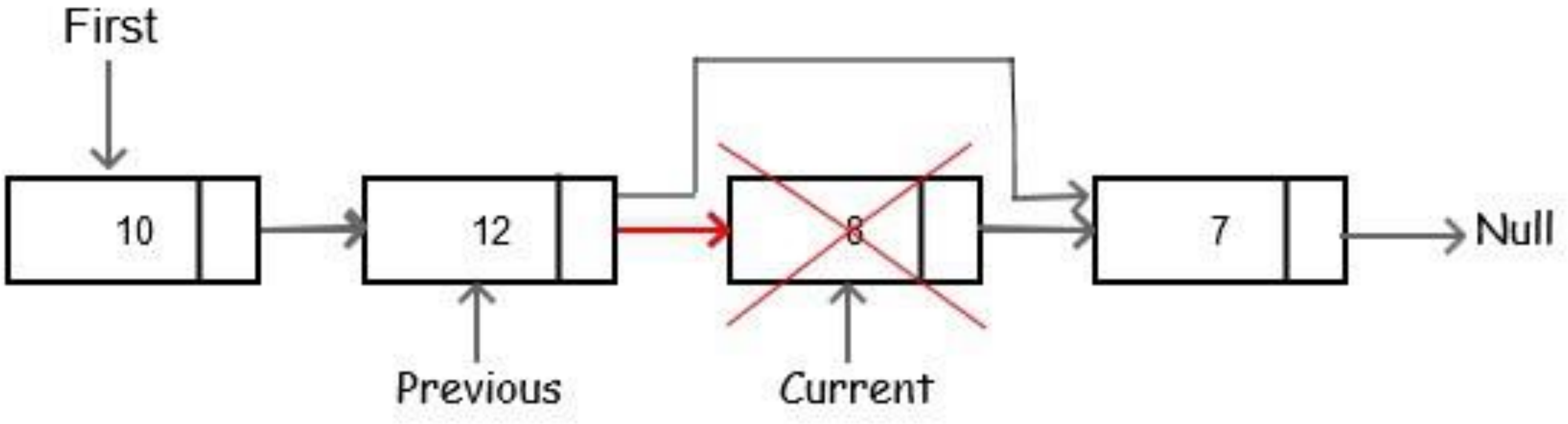
Inserting and Deleting

by list traversal



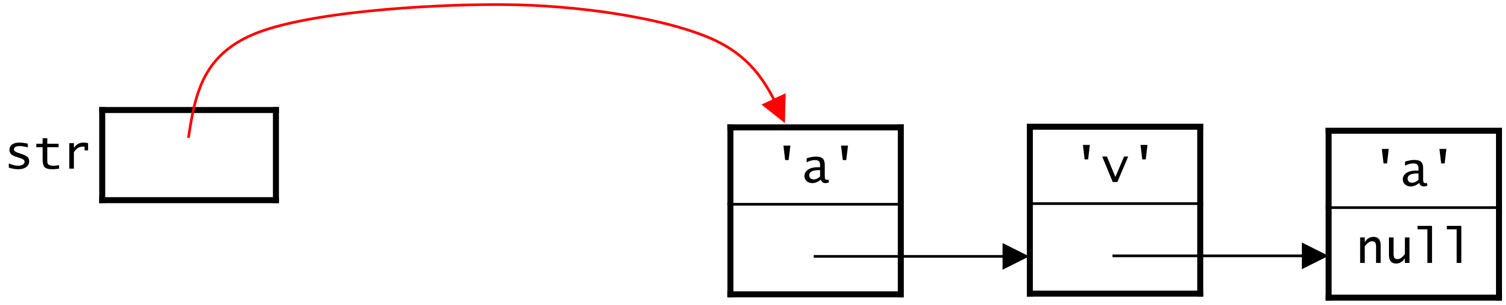
Node to Delete = 8

Note

Deleting the Item at Position i

- Special case: `i == 0` (deleting the first item)
- Update our reference to the first node:

```
    str = str.next;
```

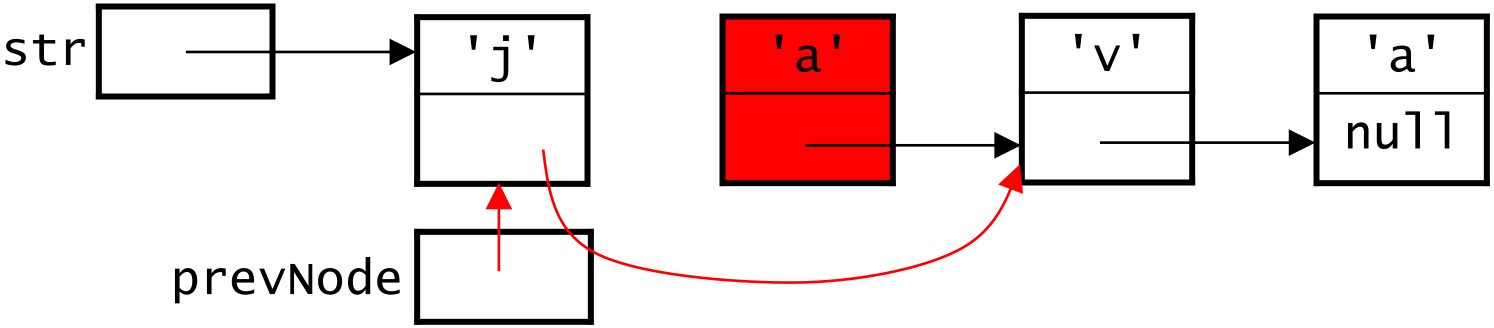


Deleting the Item at Position i (cont.)

- <u>General case:</u> `i > 0`
- First obtain a reference to the *previous* node:

```
StringNode prevNode = getNode(i - 1);
```

(example for i == 1*)*



- `prevNode.next = prevNode.next.next;`

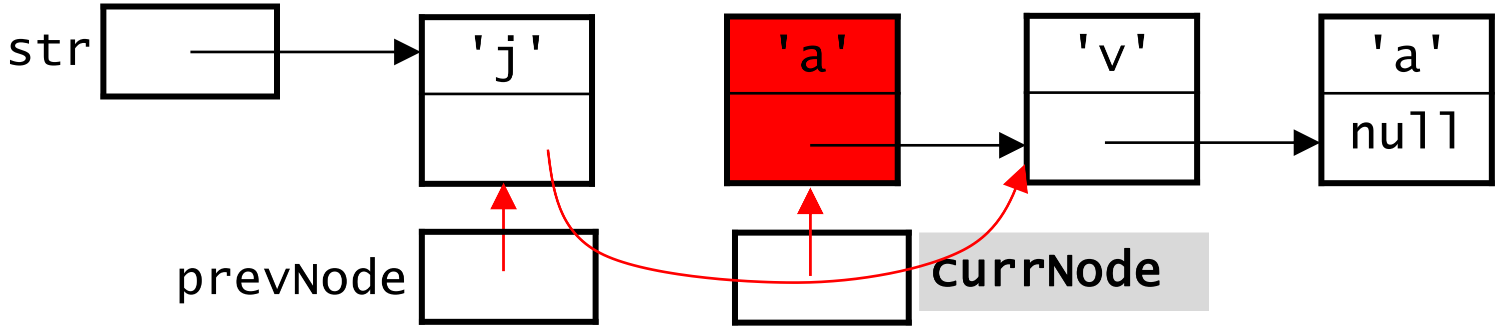
Deleting the Item at Position i

(an alternative, **use a second reference**)

- <u>General case:</u> `i > 0`
- Also obtain a reference to the node being deleted:

```
StringNode currNode = getNode(i);
```

(example for i == 1*)*



- `prevNode.next = currNode.next;`

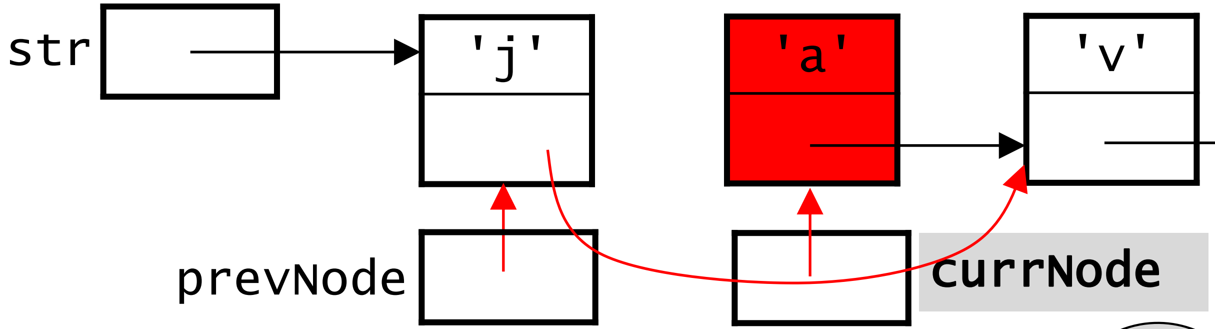
Deleting the Item at Position i

(an alternative, **use a second reference**)

- <u>General case:</u> `i > 0`
- Also obtain a reference to the node being deleted:

```
StringNode currNode = getNode(i);
```

(example for i == 1*)*



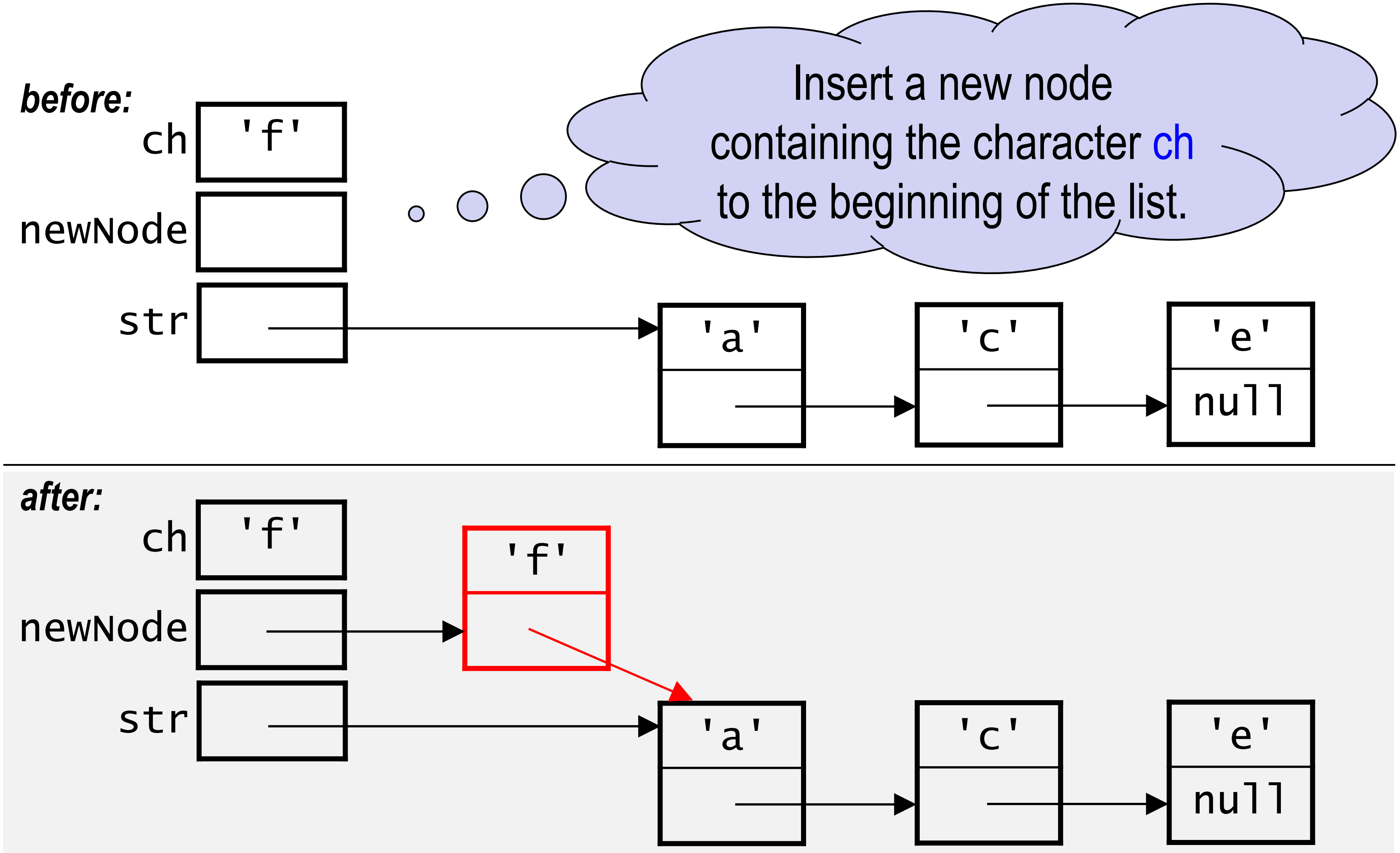
- `prevNode.next = currNode.next;`

Note that to establish the two references, prevNode and currNode we called the method, `getNode` twice:

`getNode(i-1)`
`getNode(i)`

Inserting an Item at Position i

- Special case: `i == 0` (insertion at the front of the list)

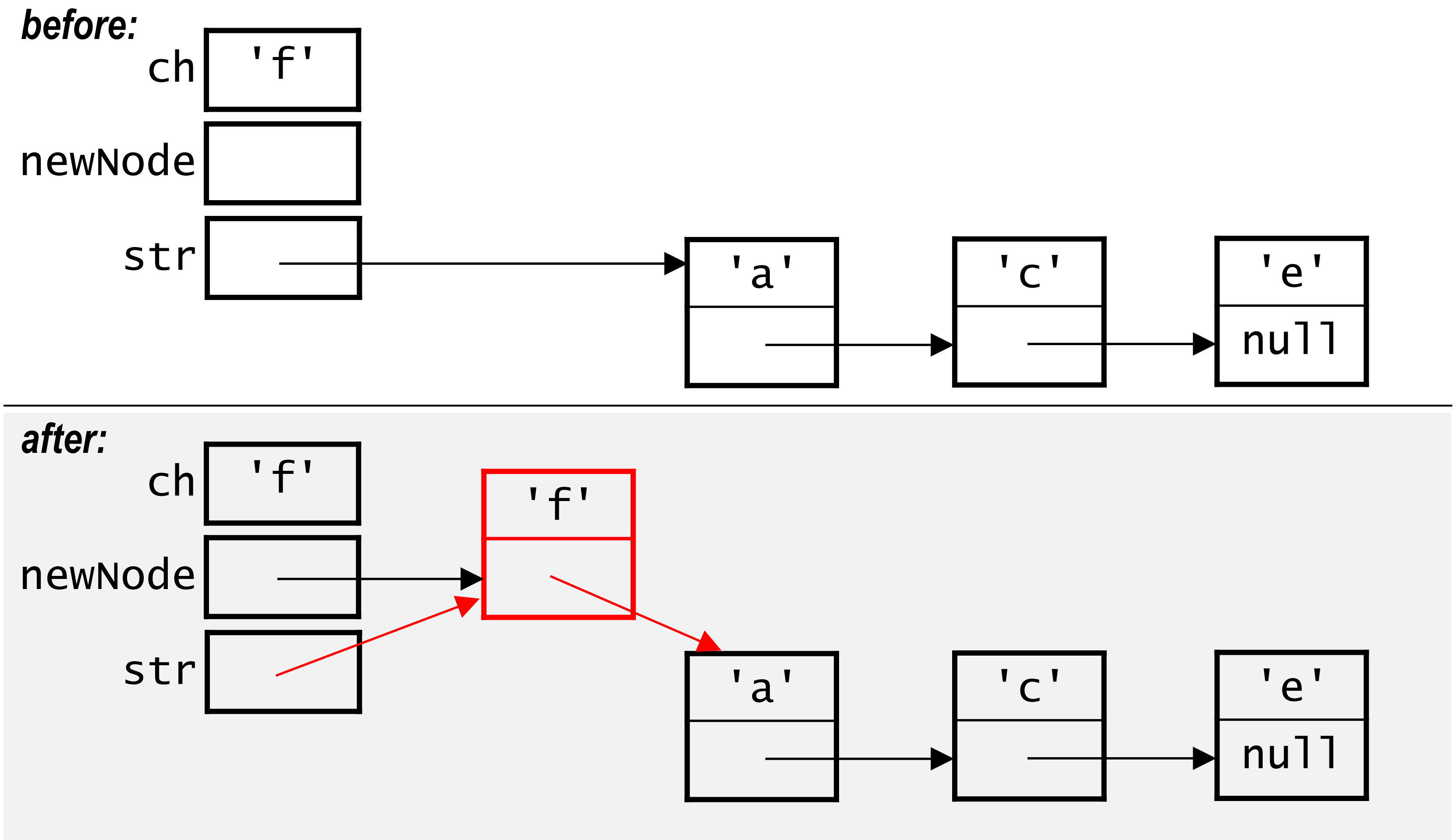


Insert a new node containing the character ch to the beginning of the list.

```
StringNode newNode = new StringNode( ch, str );
```

Inserting an Item at Position i

- Special case: `i == 0` (insertion at the front of the list)

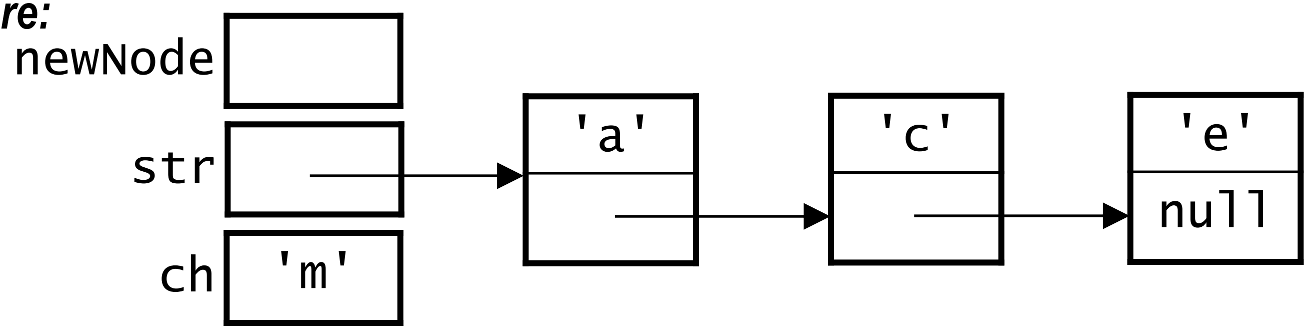


```
str = newNode;
```

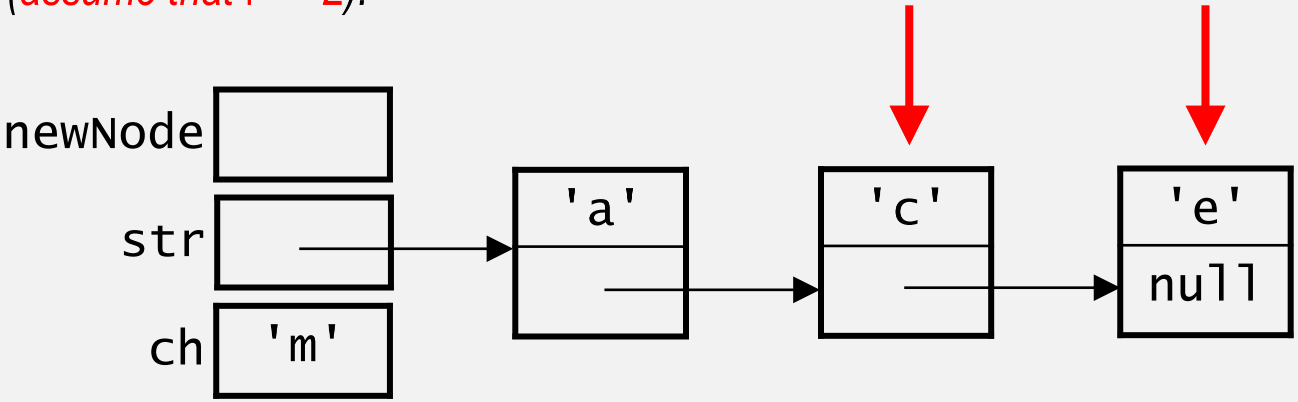
Inserting an Item at Position i (cont.)

- General case: `i > 0` (insert *before* the item currently in posn `i`)

before:



after *(assume that i == 2):*



Statement to find the node at position i-1?

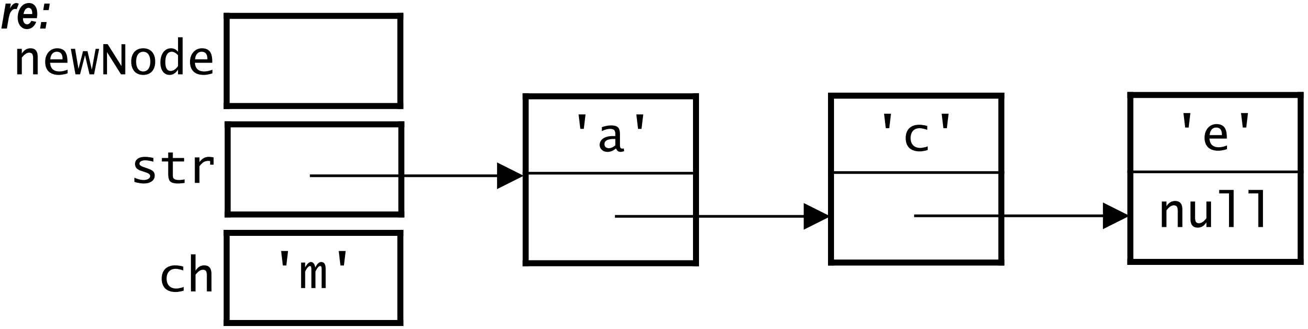
Statement to create the new node?

Assignment statement that inserts the node in the list?

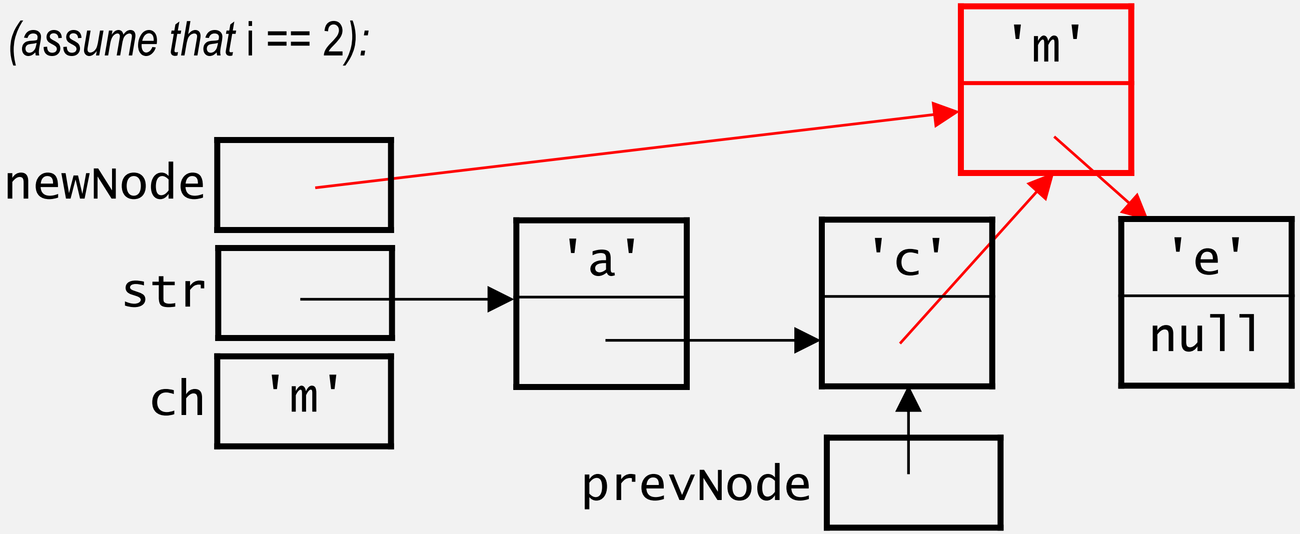
Inserting an Item at Position i (cont.)

- <u>General case:</u> `i > 0` (insert *before* the item currently in posn `i`)

before:



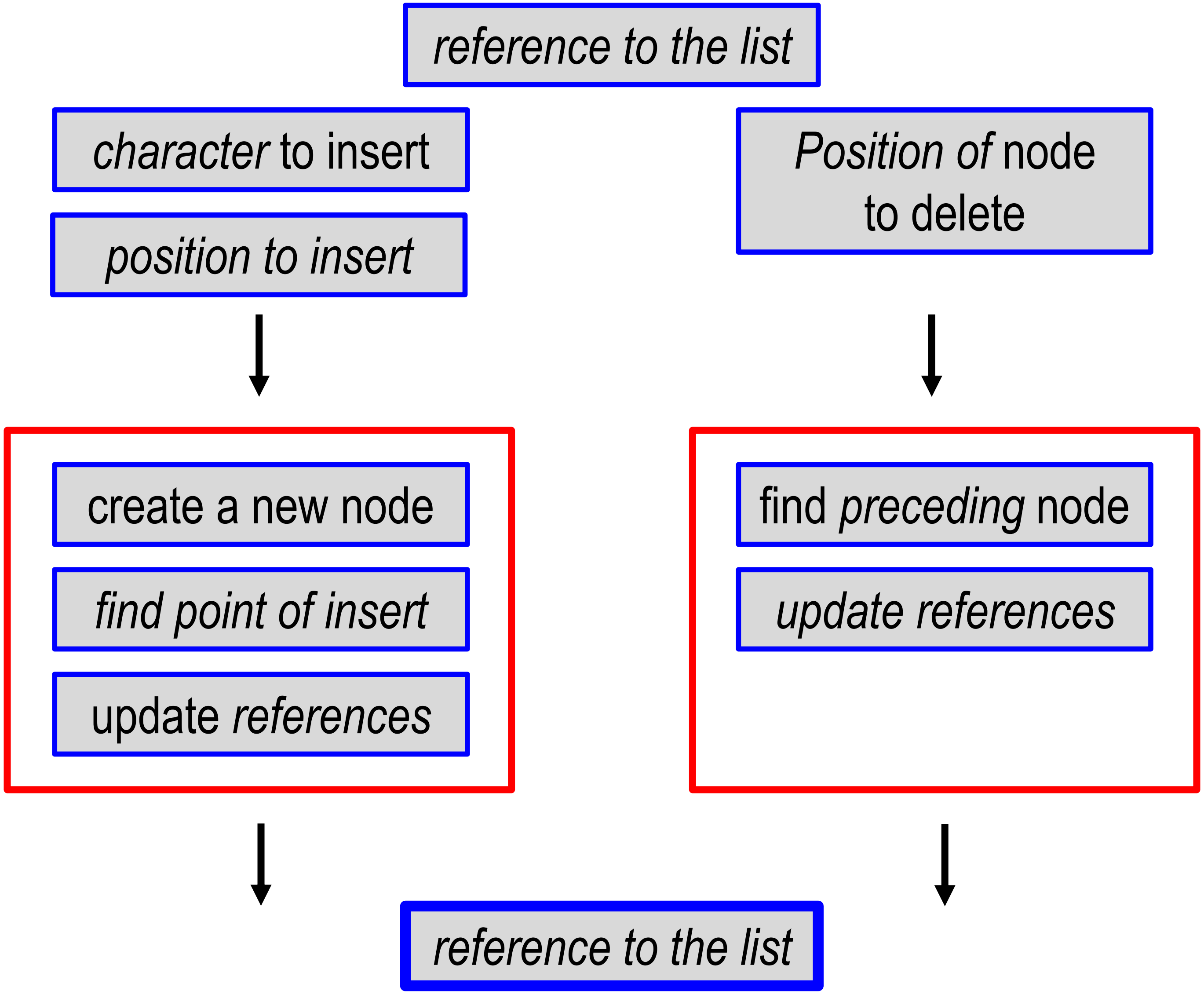
after *(assume that i == 2):*



```
StringNode prevNode = getNode(i - 1);
StringNode newNode = new StringNode(ch, prevNode.next);
prevNode.next = newNode;
```

Methods to insert and delete:

into a StringNode list



reference to the list

character to insert

position to insert

Position of node to delete

create a new node

find point of insert

update *references*

find *preceding* node

update references

reference to the list

Returning a Reference to the First Node

- Both `deleteChar()` and `insertChar()` return a reference to the first node in the linked list. For example:

```
private static StringNode deleteChar(StringNode str, int i) {
    …
    if (i == 0)                  // case 1
        str = str.next;
    else {                       // case 2
        StringNode prevNode = getNode(str, i-1);
        if (prevNode != null && prevNode.next != null)
            prevNode.next = prevNode.next.next;
        …
    }

    return str;
}
```

- The first node of the list may change.

Returning a Reference to the First Node

- Both `deleteChar()` and `insertChar()` return a reference to the first node in the linked list. For example:

```
private static StringNode deleteChar(StringNode str, int i) {
    …
    if (i == 0)                         // case 1
        str = str.next;
    else {                              // case 2
        StringNode prevNode = getNode(str, i-1);
        if (prevNode != null && prevNode.next != null)
            prevNode.next = prevNode.next.next;
        …
    }

    return str;
}
```

- The first node of the list may change.
- Invoke as follows: **`str = StringNode.deleteChar(str, i);`**
  `str = StringNode.insertChar(str, i, ch);`
- If the first node changes, `str` will point to the new first node.

Returning a Reference to the First Node

- Both `deleteChar()` and `insertChar()` return a reference to the first node in the linked list. For example:

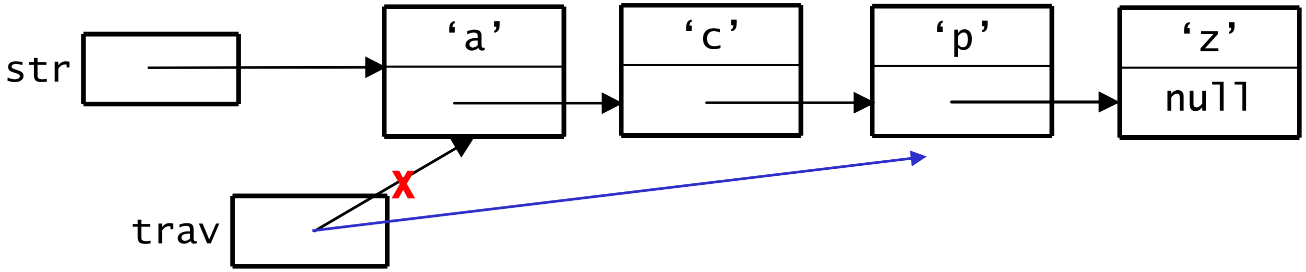
```
private static StringNod                    ode str, int i) {
    …
    if (i == 0)
        str = str.nex
    else {
        StringNode pr                                   );
        if (prevNode                                  != null)
            prevNode.                               ext;
        …
    }

    return str;
}
```

What if we did not know where in the list to insert the new node? Say we wanted to insert based on some order…

- The first node of the list may change.
- Invoke as follows: `str = StringNode.deleteChar(str, i);`
  `str = StringNode.insertChar(str, i, ch);`
- If the first node changes, `str` will point to the new first node.

Using a "Trailing Reference" During Traversal

- When traversing a linked list, using a single `trav` reference isn't always good enough.
- Ex: insert `ch = 'n'` at the right place in this *sorted* linked list:



- Traverse the list to find the right position:

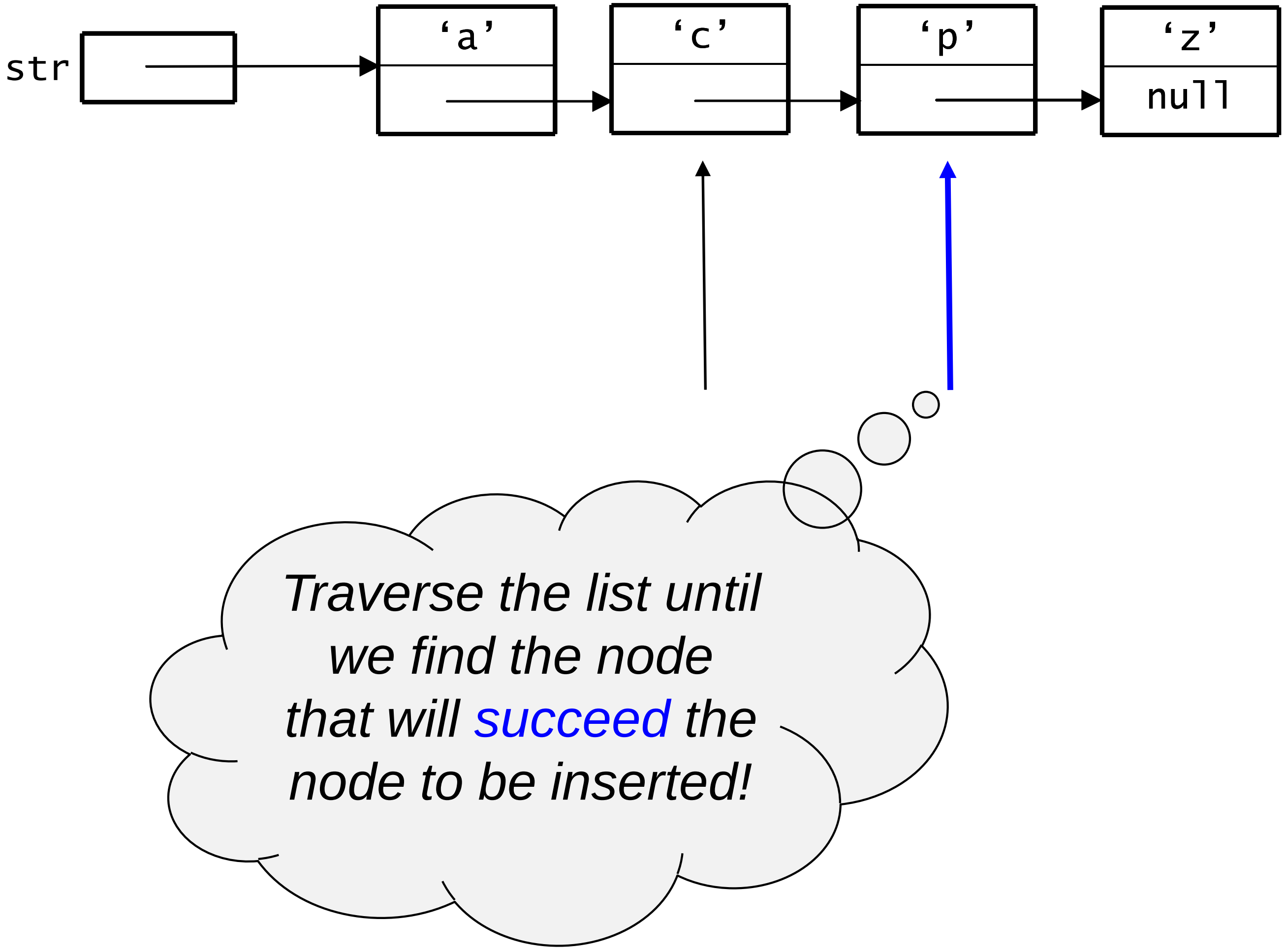
```
StringNode trav = str;
while (trav != null && trav.ch < ch)
    trav = trav.next;
```

- When we exit the loop, where will `trav` point? Can we insert `'n'`?
- No, we need a reference to the previous node!

Inserting and Deleting

by trailing reference

- Ex: insert `ch = 'n'` at the right place in this *sorted* linked list:

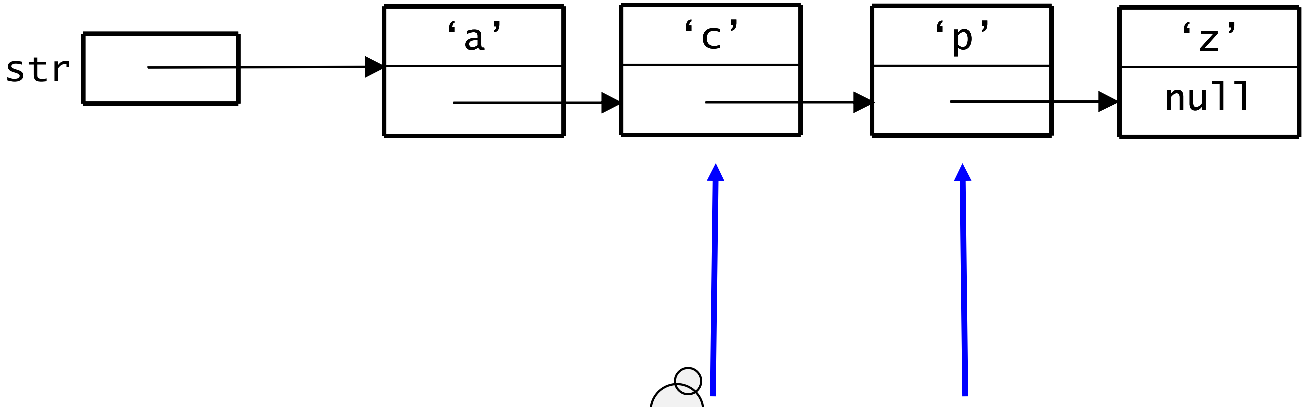


Traverse the list until we find the node that will succeed the node to be inserted!

Inserting and Deleting

by trailing reference

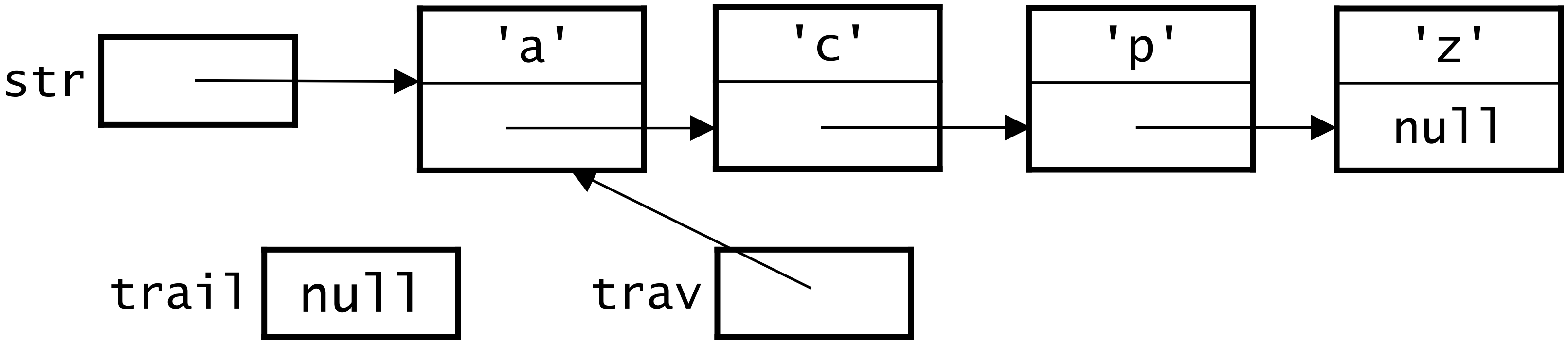
- Ex: insert `ch = 'n'` at the right place in this *sorted* linked list:



But also keep track of the node that will precede the node to be inserted!

Using a "Trailing Reference" (cont.)

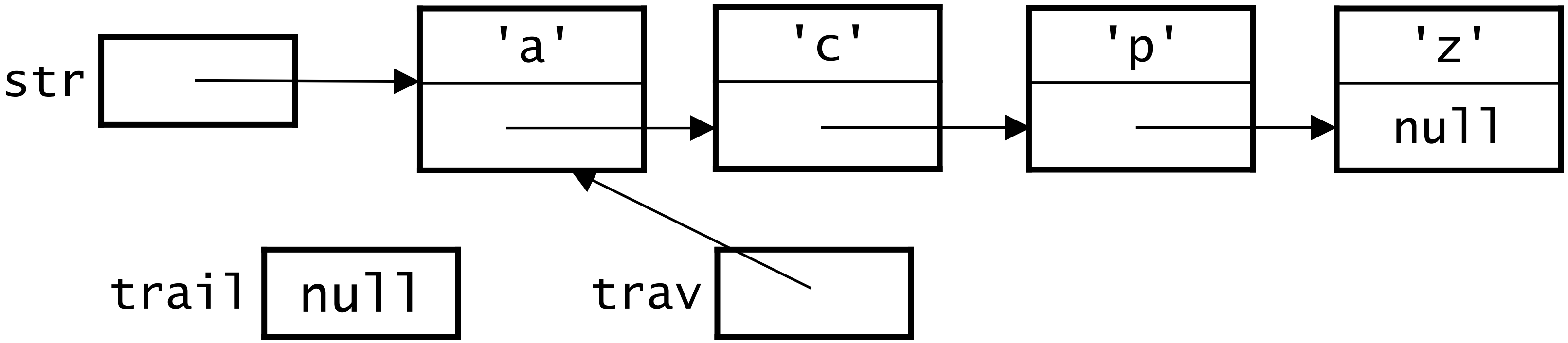
- To get around the problem, traverse the list using two different references:
  - `trav`, which we use as we did before
  - `trail`, which stays one node behind `trav`



```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

Using a "Trailing Reference" (cont.)

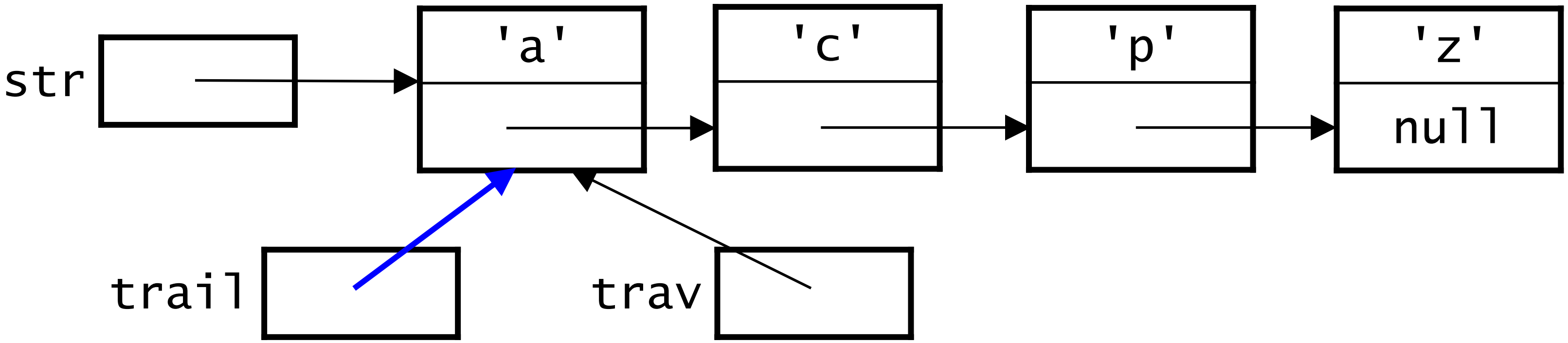
- To get around the problem, traverse the list using two different references:
    - `trav`, which we use as we did before
    - `trail`, which stays one node behind `trav`



```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

Using a "Trailing Reference" (cont.)

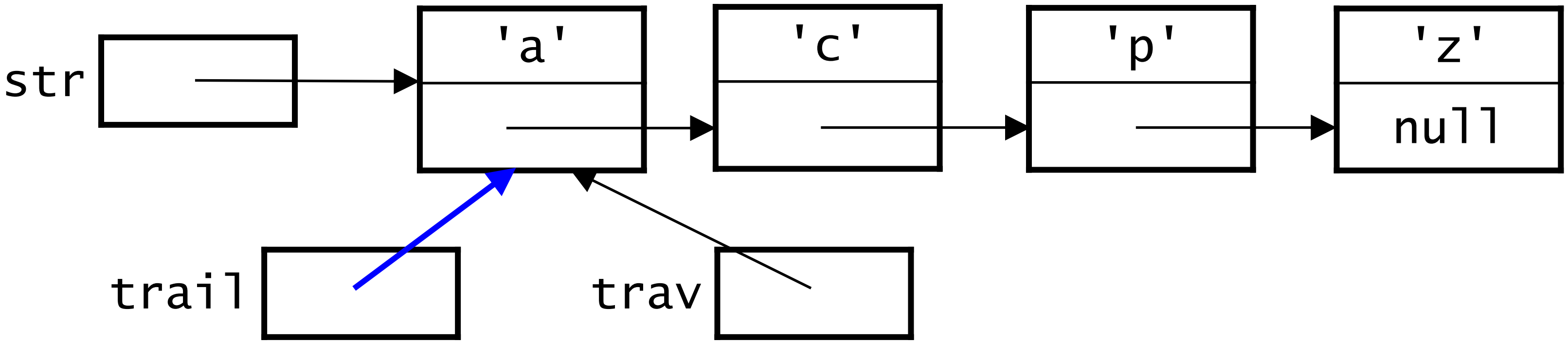
- To get around the problem, traverse the list using two different references:
  - `trav`, which we use as we did before
  - `trail`, which stays one node behind `trav`



```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

Using a "Trailing Reference" (cont.)

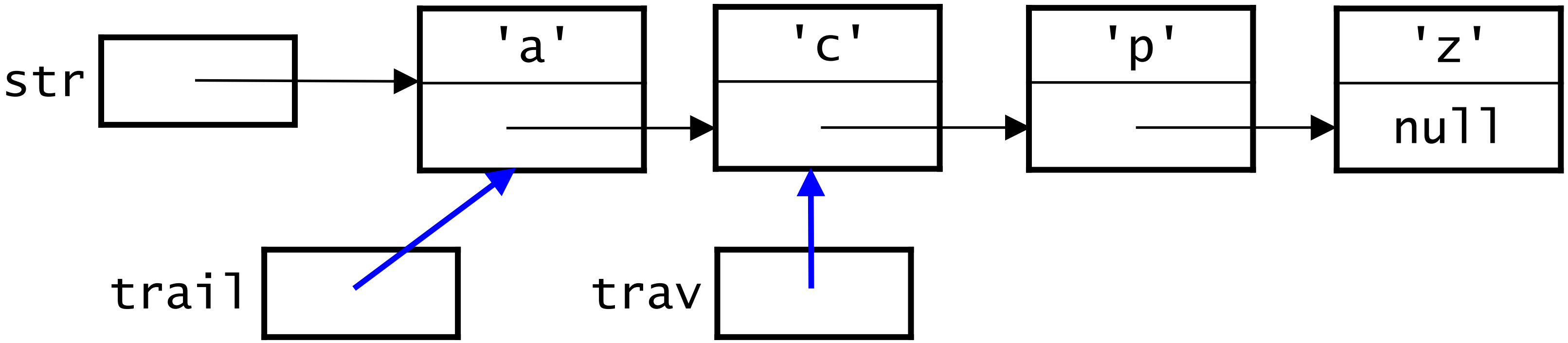
- To get around the problem, traverse the list using two different references:
  - `trav`, which we use as we did before
  - `trail`, which stays one node behind `trav`



```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

Using a "Trailing Reference" (cont.)

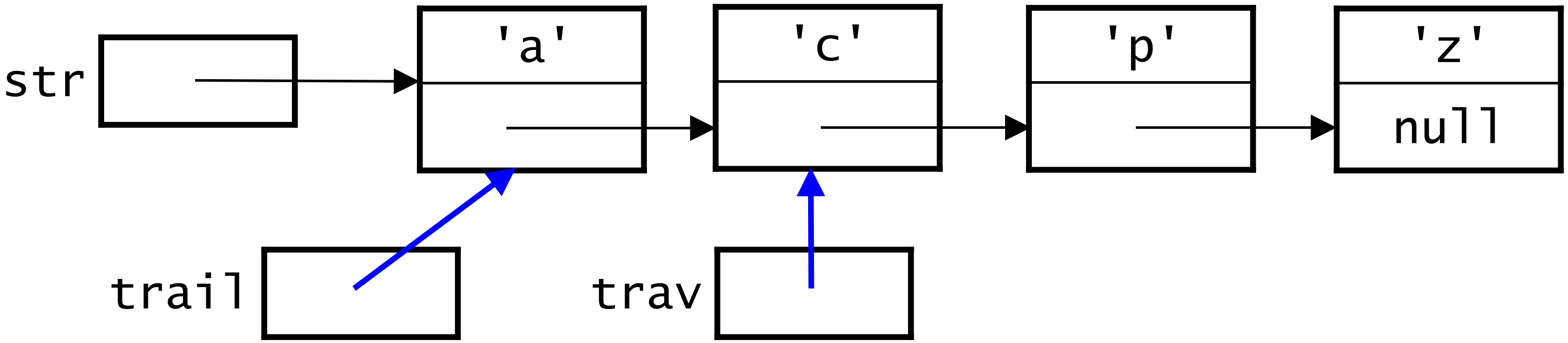
- To get around the problem, traverse the list using two different references:
  - `trav`, which we use as we did before
  - `trail`, which stays one node behind `trav`



```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

Using a "Trailing Reference" (cont.)

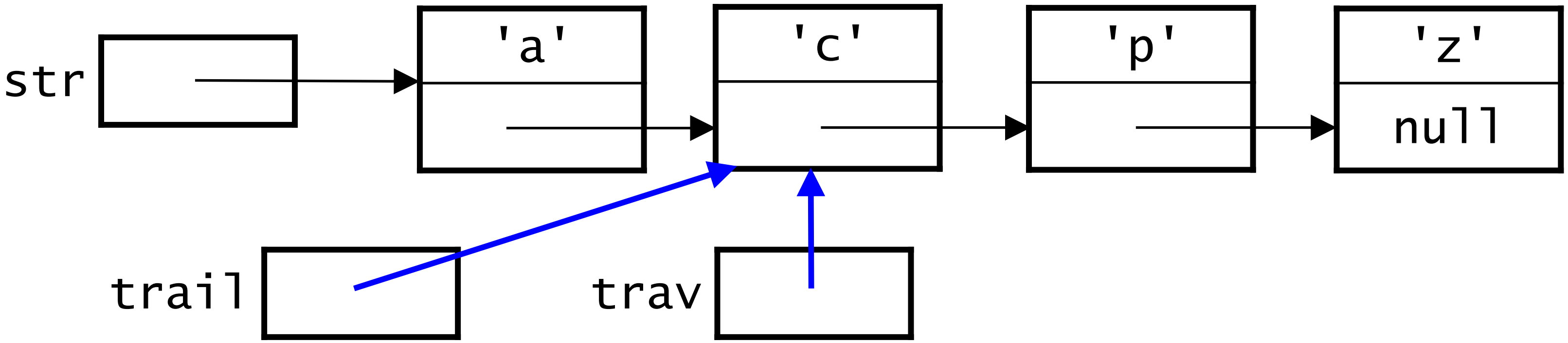
- To get around the problem, traverse the list using two different references:
  - `trav`, which we use as we did before
  - `trail`, which stays one node behind `trav`



```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

Using a "Trailing Reference" (cont.)

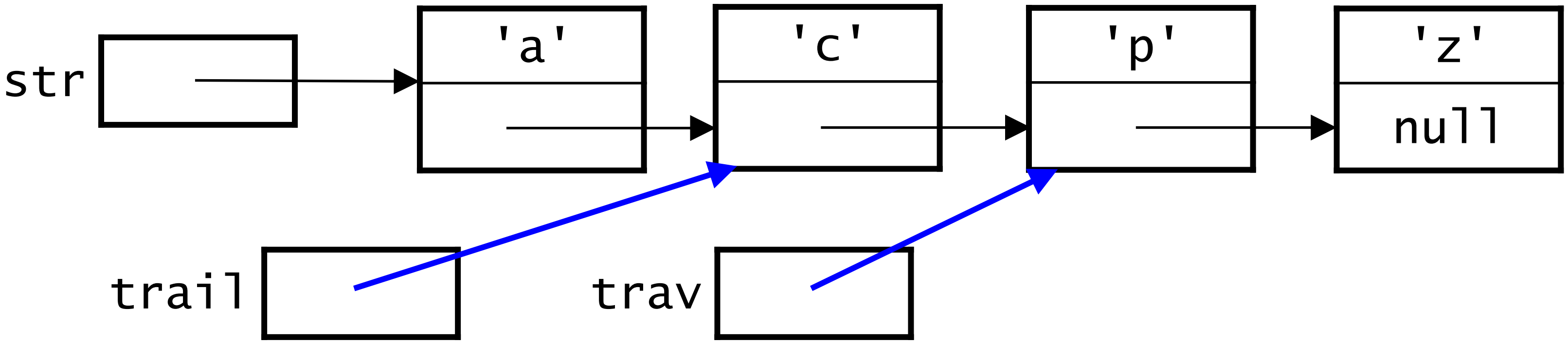
- To get around the problem, traverse the list using two different references:
  - `trav`, which we use as we did before
  - `trail`, which stays one node behind `trav`



```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

Using a "Trailing Reference" (cont.)

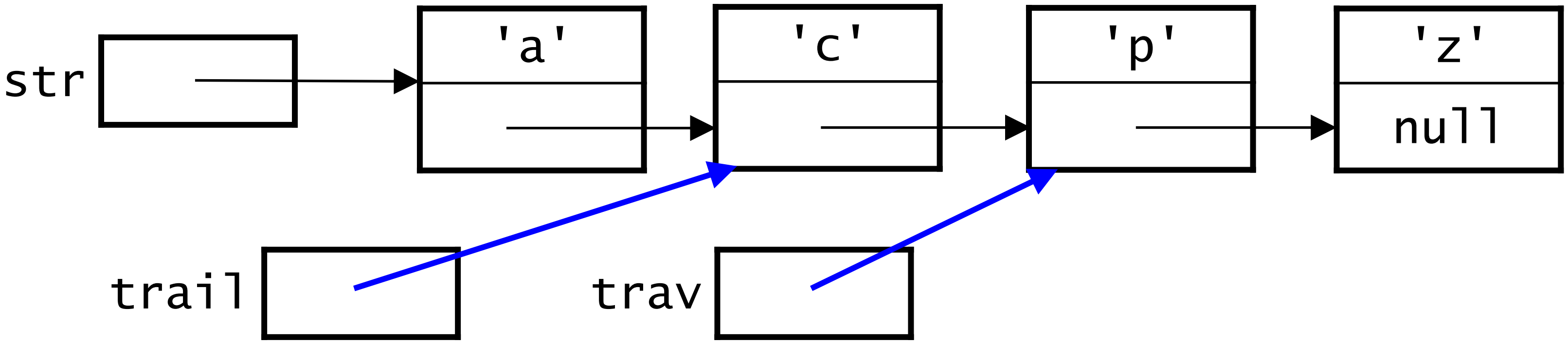
- To get around the problem, traverse the list using two different references:
  - `trav`, which we use as we did before
  - `trail`, which stays one node behind `trav`



```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

Using a "Trailing Reference" (cont.)

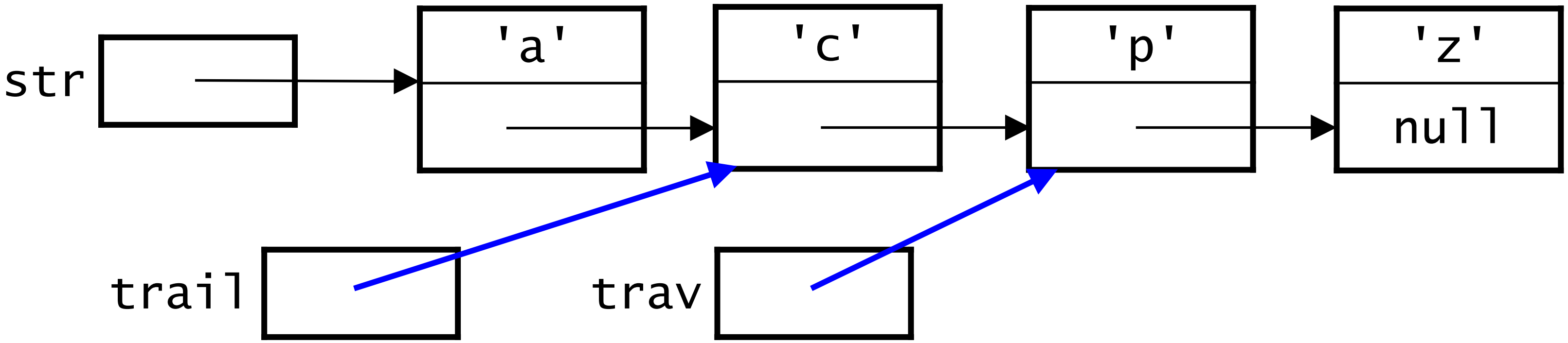
- To get around the problem, traverse the list using two different references:
  - trav, which we use as we did before
  - trail, which stays one node behind trav



```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

Using a "Trailing Reference" (cont.)

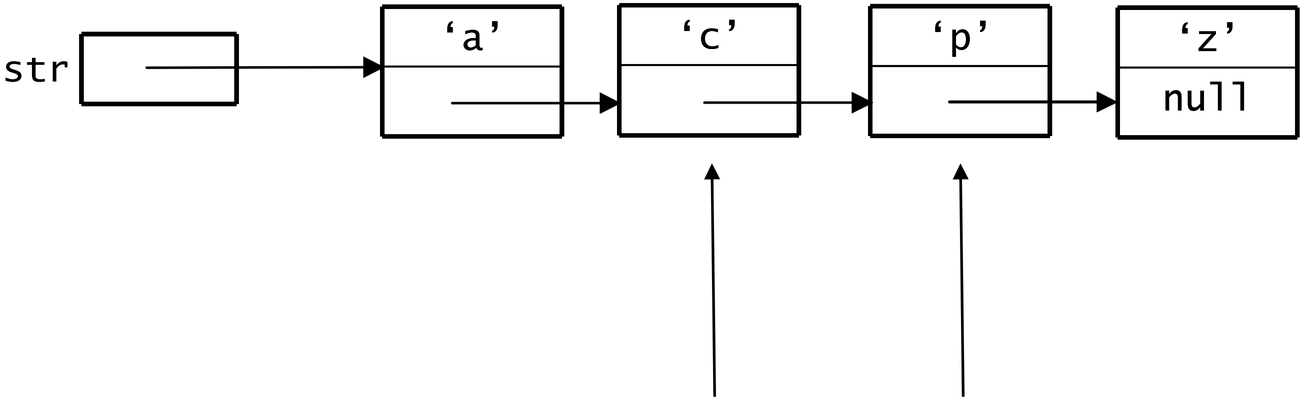
- To get around the problem, traverse the list using two different references:
  - `trav`, which we use as we did before
  - `trail`, which stays one node behind `trav`



```
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

Inserting and Deleting

by trailing reference



Key to *inserting* a new node in the list and *deleting* an existing node in the list using list traversal (in a single linked list) is to maintain two references to the list:

- trail – a reference to the previous node (or trailing node)
- trav – a reference to the current node

Deleting a node in the list:

establishing two references

- `deleteChar(str, ch)` – a private method that deletes from the list the *first* node containing the character ch.

- Two references approach:

```
private static StringNode deleteChar(StringNode str, Char ch) {
    StringNode trav = str, trail = null;
    // traverse until the desired node is found
    while( trav != null && trav.item != ch ) {
        trail = trav;
        trav = trav.next;
    }
    if ( trav != null ) {
        // node to delete has been found
        if ( trail != null )
            // not deleting the first node
            trail.next = trav.next;
        else
            // deleting the first node – reassign str
            str = trav.next;
    }
    return( str );
}
```

Deleting *multiple* nodes in the list:

establishing two references

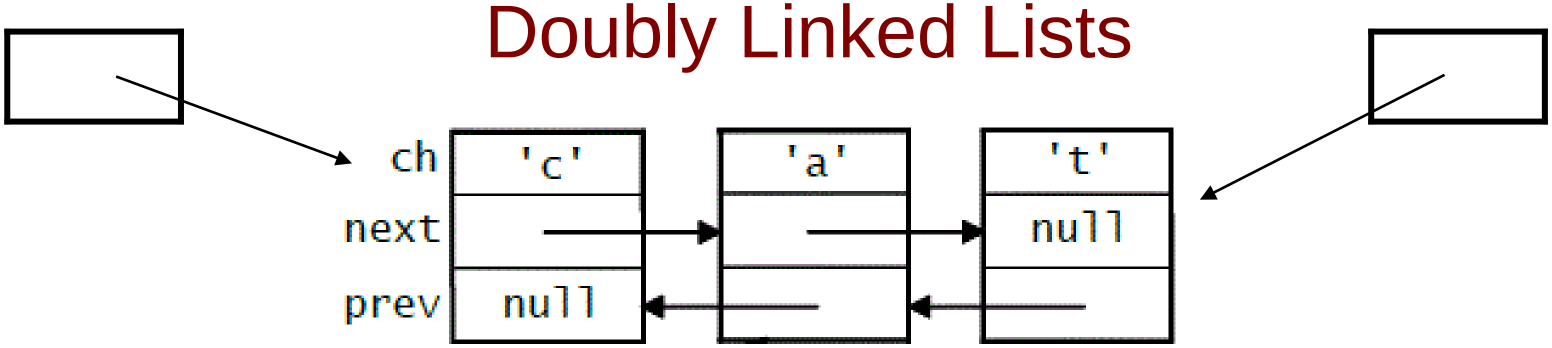
- `deleteAllChar(str, ch)` – a private method that deletes from the list the *all* nodes containing the character ch.

- Two references approach:

```
 private static int deleteAllChar(StringNode str, Char ch) {
```

```
}
```

Doubly Linked Lists



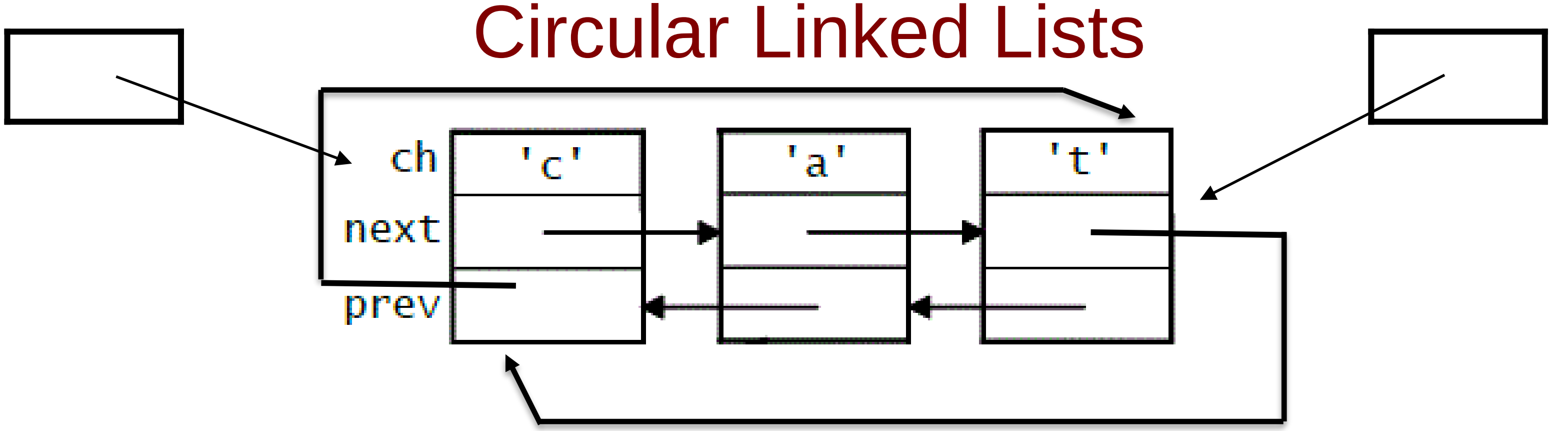
- In a doubly linked list, every node stores *two* references:
  - `next`, which works the same as before
  - `prev`, which holds a reference to the previous node
    - in the first node, `prev` a value of `null`

- The `prev` references allow us to "back up" as needed.
  - remove the need for a trailing reference during traversal!

- Insertion and deletion must update both types of references.

- Allows us to maintain a tail reference to the last node in the list!

Circular Linked Lists



- In a circular linked list, the next pointer of the last node references the first node in the list:

- In a circular double linked list, the *prev* pointer of the first node in the list references the last node in the list.