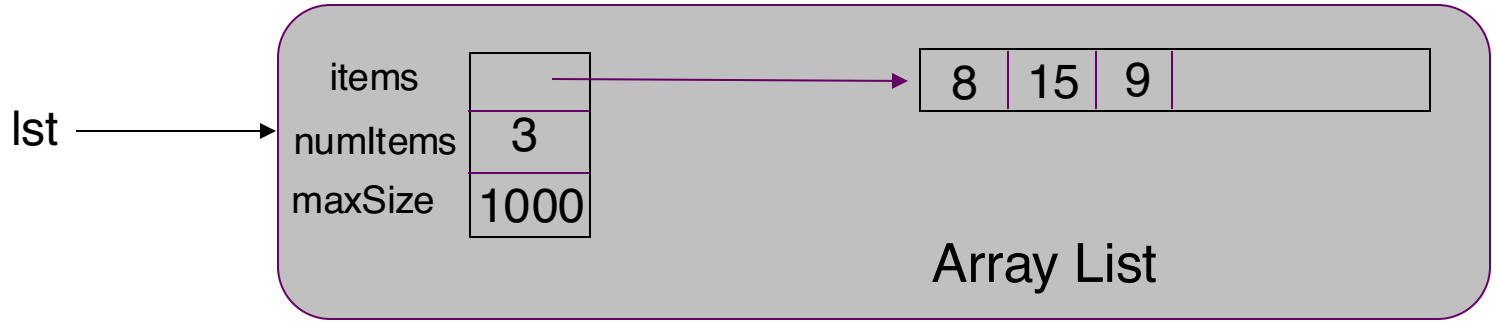
Lists

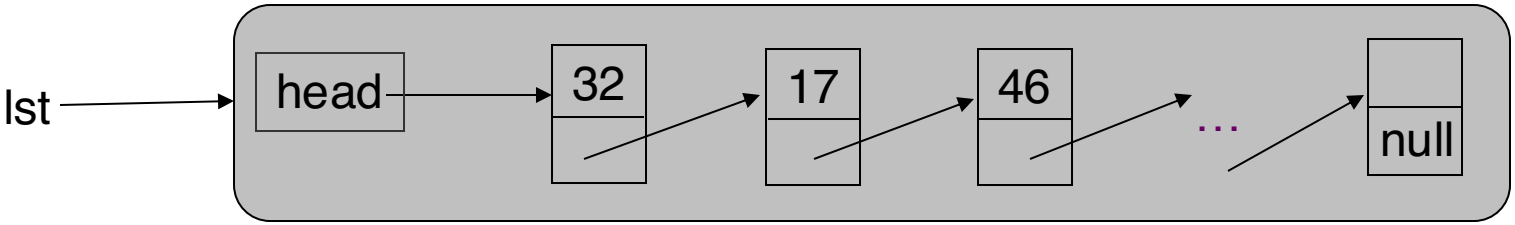
EECS 233

Array Representation



- Store list elements wall-to-wall in memory in an array
- Keep a variable recording the current number of elements
- Advantages
  - Easy and efficient access to *any* item in the sequence
    - items[i] gives you the item at position i
    - Random access
      - Every item can be accessed in constant time given its index
  - Very compact: no auxiliary fields are required
- Disadvantages of using an array:
  - The need to specify an initial array size and resize as required (how?)
  - Difficult to insert/delete items at arbitrary positions (running time?)
  - May have many empty positions

Linked List Representation

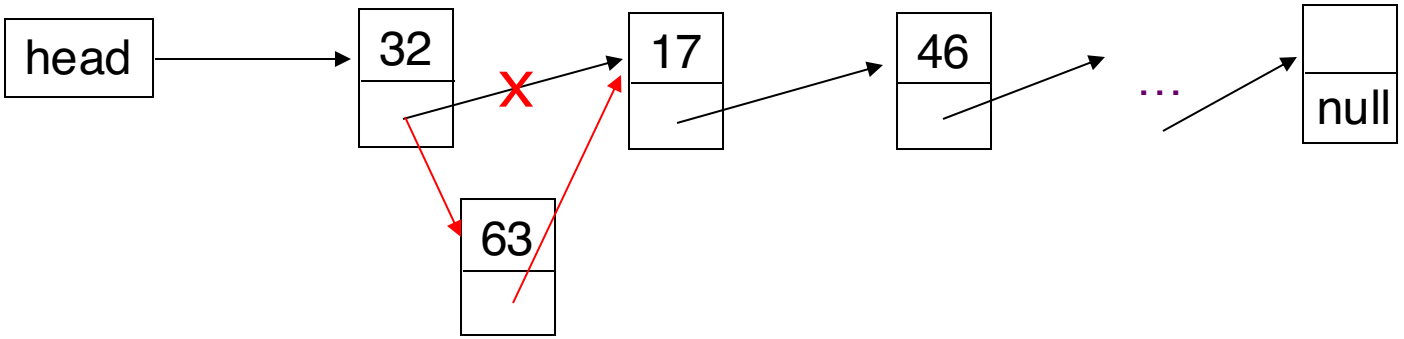


- More efficient data structure for a dynamic sequence (with frequent insert/delete operations)
- A linked list stores a sequence of items in separate *nodes*. Each node contains:
  - a single item, and
  - a "link" (i.e., a reference/pointer) to the node containing the next item

  The last node in the linked list has a link value of NULL or null.
- The linked list starts with a variable that holds a reference to the first node – head of the list

Advantages/Disadvantages of Linked List

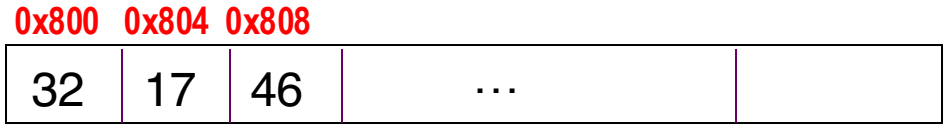
- Advantages:
  - No capacity limit (provided there is enough memory).
  - Easy to insert/delete an item – no need to "shift over" other items.
    - Done in constant time.



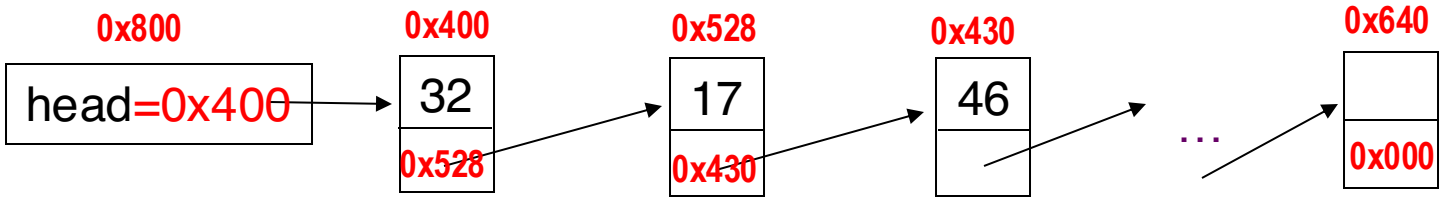
- Disadvantages:
  - No random access
    - "walk down" the list to access an item
  - Memory overhead for the links

Memory Management for Linked Lists

- In an array, the elements occupy consecutive memory locations in the heap:
  - Address in red

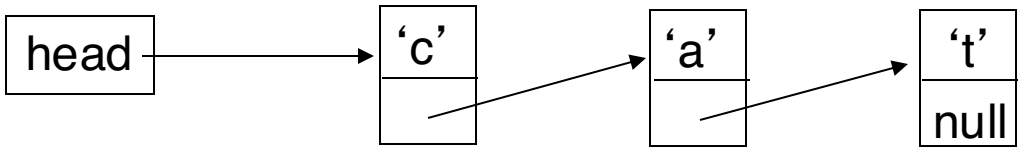


- In a linked list, each node is a distinct object in the heap. The nodes do *not* have to be next to each other in memory.



An Example Linked List

- A string represented using a linked list: **LLString**. Each node in the linked list represents one character.

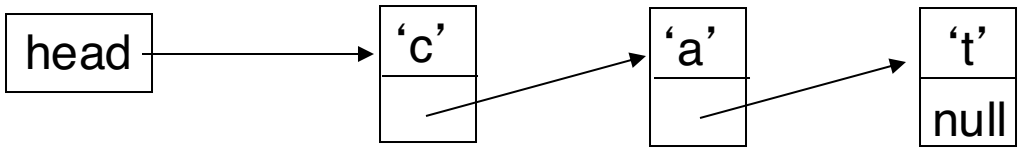


- Java classes

```
public class StringNode {
        private char ch;
        private StringNode next;
        …
}
```

```
public class LLString {
        private StringNode head;
        private int theSize;
        …
```

Under the Hood of the String Linked List



- The string as a whole will be represented *(internally)* by a variable that holds a reference to the node containing the first character.

  StringNode str1;

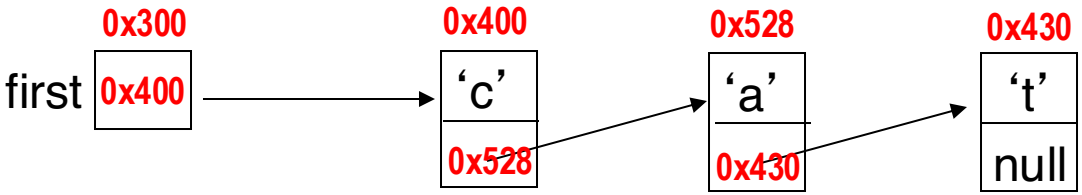
- An empty string will be represented by a NULL value.

  StringNode str2 = null;

- We will use helper methods that take the first node of the string as a parameter.
  - We will have length(str1) instead of str1.length()
  - This is necessary so that the methods can handle empty strings.
    - if str1 == NULL, length(str1) will work, but str1.length() will produce a runtime error

More on References

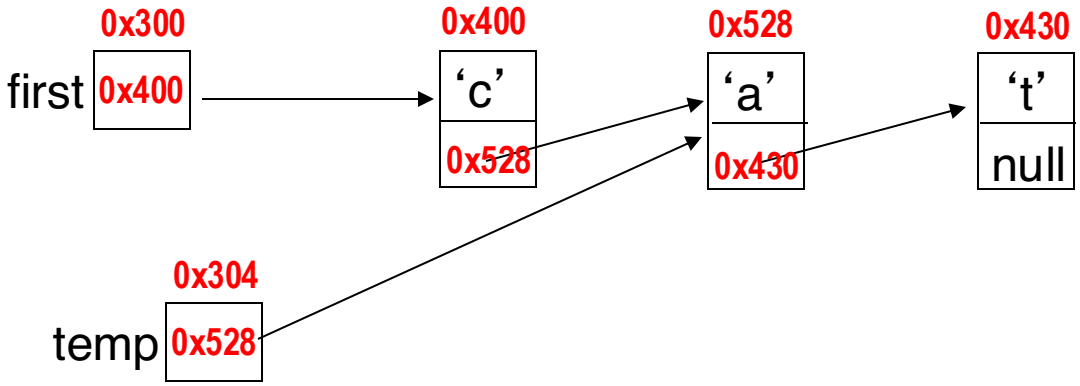
- A reference is also a variable
  - that has its location in the memory, and
  - whose value is the address (i.e., location) of data



  - e.g., first: address=0x300, value=0x400
  - How about first.next.next ?
  - How about first.next.ch ?

More on References

- Example: temp.next.ch



- Start with expression: "**temp**." It represents the node to which **temp** refers.
- Now consider "**temp.next**". It represents the field "next" of the node to which temp refers.
  - address = ?
  - value = ?
- Next, consider "**temp.next.**". It represents the node with address 0x430.
- Finally, **temp.next.ch** represents 't'.

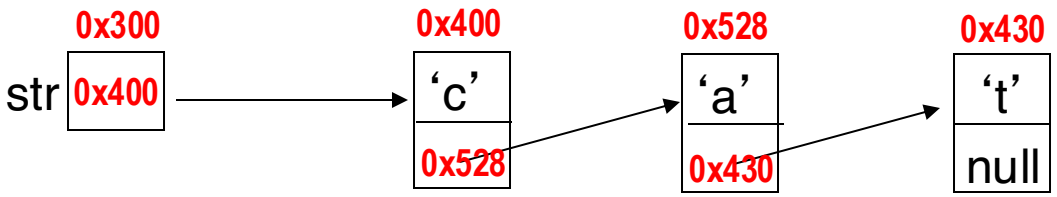
Recursion on Linked Lists

- Recursive definition of a linked list: a linked list is either
  - empty or
  - a single node, followed by a linked list

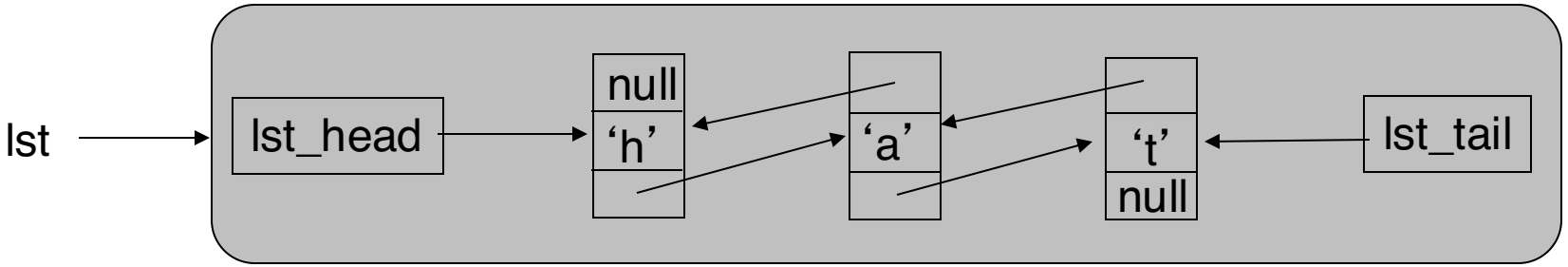
- Recursive definition lends itself to recursive methods.

- Example: length of a string
  - length of "cat" = 1 + length of "at"
  - length of "at" = 1 + length of "t"
  - length of "t" = 1 + length of the empty string (which is 0)

```
private static int length(StringNode str) {
    if (str == null)
        return 0;
    else
        return 1 + length(str.next);
}
```



Doubly Linked List



- Both next and prev are defined in StringNode
- Why needed?

```
public class LLString {
    private StringNode lst_head;
    private StringNode lst_tail;
    private int theSize;
    …
```

```
public class StringNode {
    private char ch;
    private StringNode next;
    private StringNode prev;
    …
}
```

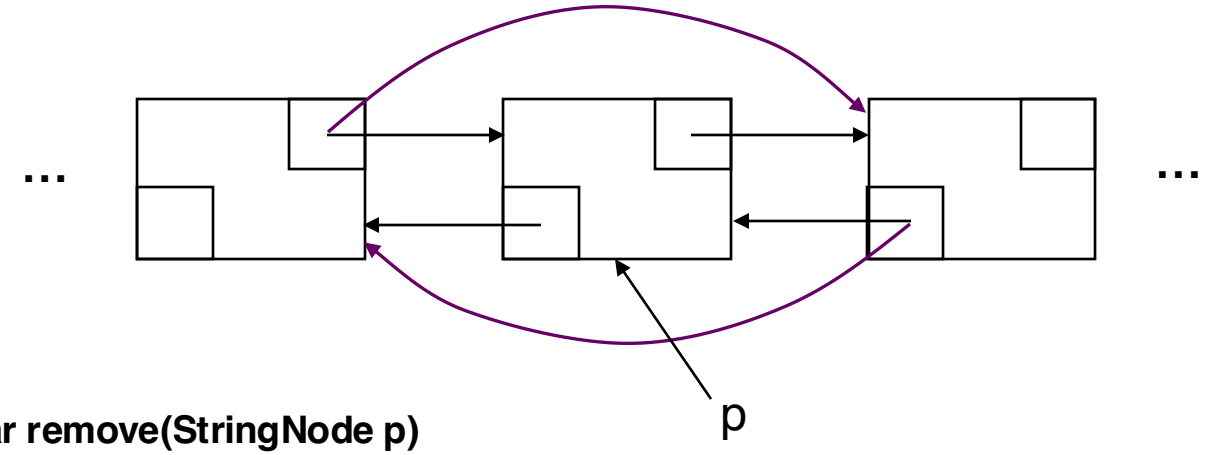
Example: Traversing Linked List

- Access the node at position *i* in a doubly linked list

```
public StringNode getNode(int i) {
        if (i < 0 || i >= theSize) throw new Exception("Index out of bounds");
        StringNode ptr;
        if (i < theSize/2) {
                ptr = lst_head;
                for (j = 0; j != i; j++) ptr = ptr.next;
        } else {
                ptr = lst_tail;
                for (j = theSize-1; j != i; j--) ptr = ptr.prev;
        }
        return ptr;
}
```

What is the running time?

Example: Removing a Node

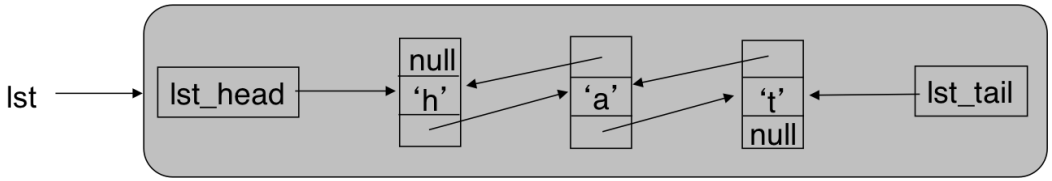



```
public char remove(StringNode p)
{
        if (p == lst_head || p == lst_tail)

                        ?

        p.next.prev = p.prev;
        p.prev.next = p.next;
        theSize--;

        return p.ch;
   }
```

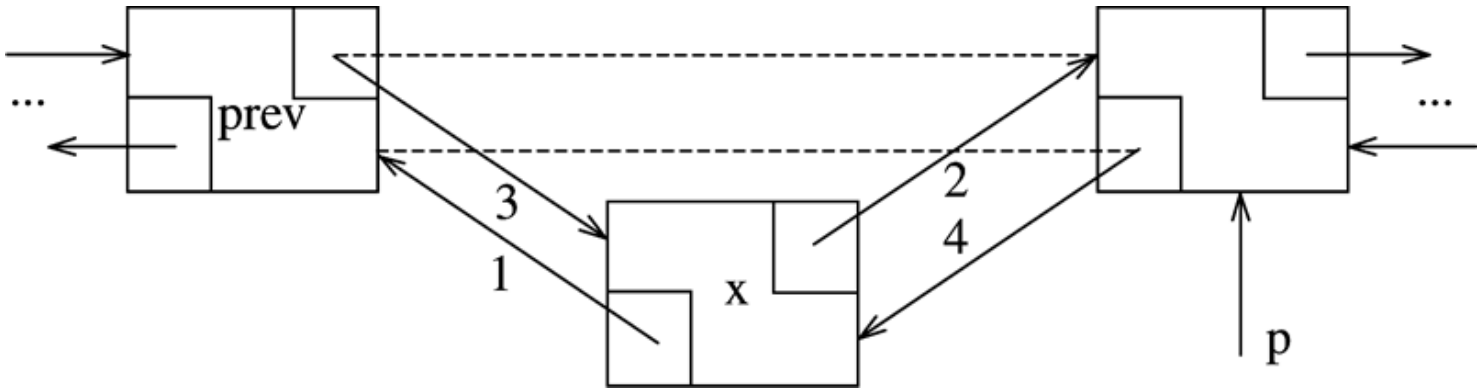


Do we need to explicitly de-allocate p?

Example: Inserting a Node

- Insert a new node before p.

What if p is the first element? Last element?
What if p == null?
What if p is not part of a list?



Other Operations

Either simple linked list or doubly linked list

- Count the occurrences of an item in the linked list
- Remove all occurrences of an item
- Reverse a linked list (trivial for doubly linked list)
- Duplicate a linked list