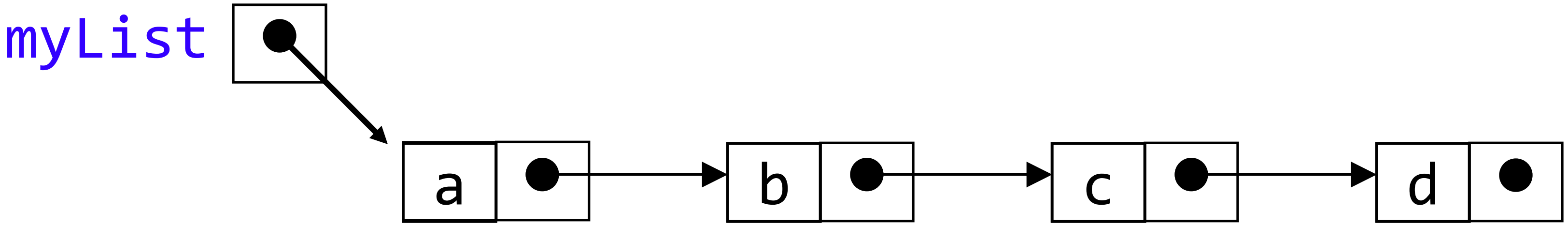
Linked Lists

Anatomy of a linked list

- A linked list consists of:
  - A sequence of nodes



Each node contains a value
and a link (pointer or reference) to some other node

The last node contains a null link

The list may (or may not) have a header

More terminology

- A node's successor is the next node in the sequence
  - The last node has no successor
- A node's predecessor is the previous node in the sequence
  - The first node has no predecessor
- A list's length is the number of elements in it
  - A list may be empty (contain no elements)

Pointers and references

- In C and C++ we have "pointers," while in Java we have "references"
  - These are essentially the same thing
    - The difference is that C and C++ allow you to modify pointers in arbitrary ways, and to point to anything
  - In Java, a reference is more of a "black box," or ADT
    - Available operations are:
      - dereference ("follow")
      - copy
      - compare for equality
    - There are constraints on what kind of thing is referenced: for example, a reference to an `array of int` can *only* refer to an `array of int`

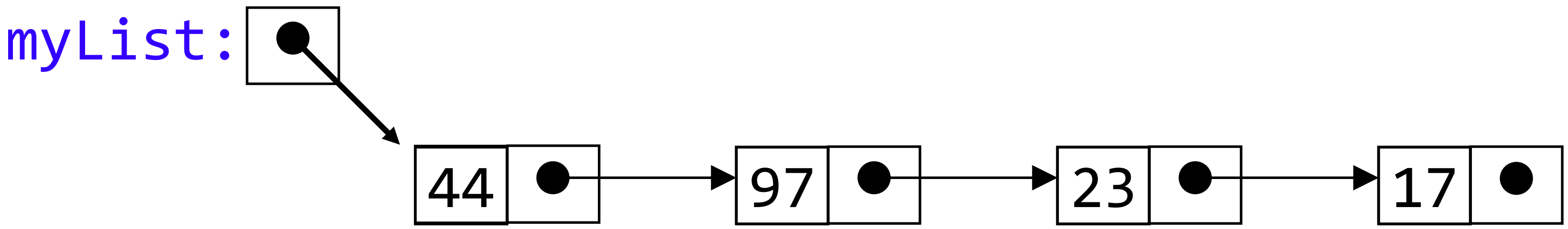
Creating references

- The keyword `malloc` creates dynamic variables, but also returns a *pointer* to that object
- Consider

```
struct node {
        struct node *next;
        int   data;
    };
typedef struct node Node;
```

- For example, `Node *p = (Node *) malloc (sizeof(Node))`

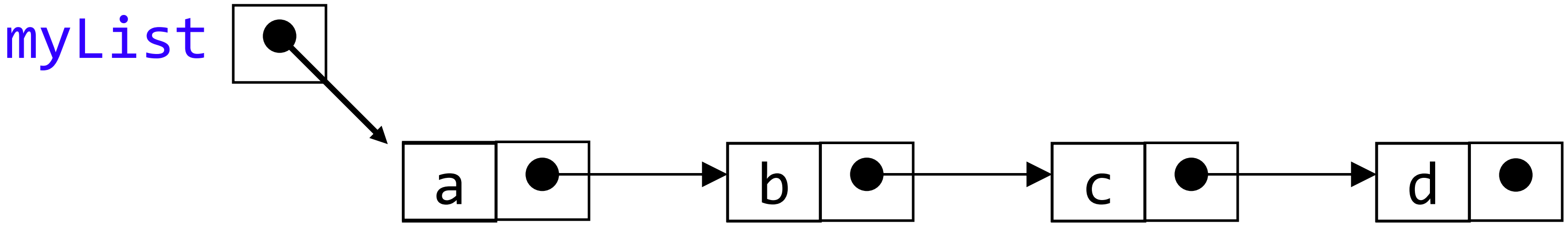
Creating link list




```
Node *temp1 = (Node *) malloc (sizeof(Node));
temp1->data = 17;
Node *temp2 = (Node *) malloc (sizeof(Node));
temp2->data = 23;
temp2->next = temp1;
Node *temp3 = (Node *) malloc (sizeof(Node));
temp3->data = 97;
temp3->next = temp2;
Node *myList = (Node *) malloc (sizeof(Node));
myList->data = 44;
myList->next = temp3;
```

Singly-linked lists

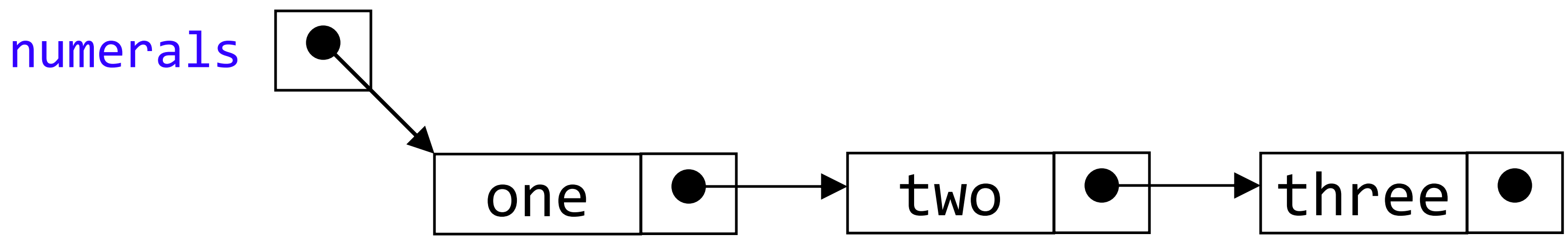
- Here is a singly-linked list (SLL):



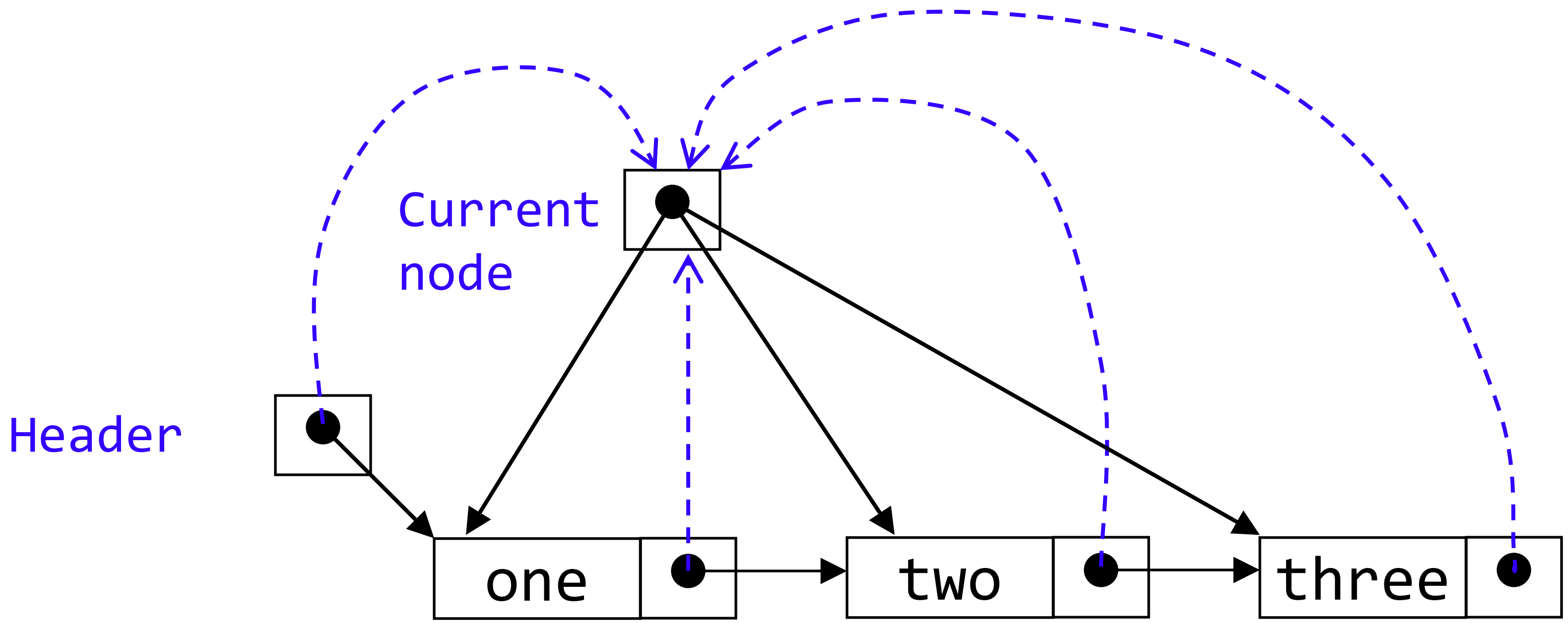
- Each node contains a value and a link to its successor (the last node has no successor)
- The header points to the first node in the list (or contains the null link if the list is empty)

Creating a simple list

- To create the list ("one", "two", "three"):
- `Node *numerals;`
- ```
  numerals =
      getnode("one",
          getnode("two",
              getnode("three", null)));
  ```


  ```

Traversing a SLL (animation)



Traversing a SLL

```
void display() {

        if Header is null
                return empty list
        else
                set current node to head
        endif
        while (current node is not end of list)
                display data of current node
                set current node to next node
        endwhile
}
```

Inserting/Deleting a node into a SLL

- Many ways to insert a new node into a list:
  - As the new first element (ins_beg)
  - As the new last element (ins_end)
  - After nth node (specified by a *position number)* Ins_pos
  - After a given value (ins_after)
  - Before a given value (ins_before)
- Delete node from a list:
  - del_beg
  - del_end
  - del_ele(value)

Inserting as a new first element

```
void ins_beg(value) {

        create a new Node
        store data in new Node
        if (list is Empty)
                set Header to new Node
        else
                new Node next points to Header node
                set Header to the new Node
        endif
}
```

Inserting as last element

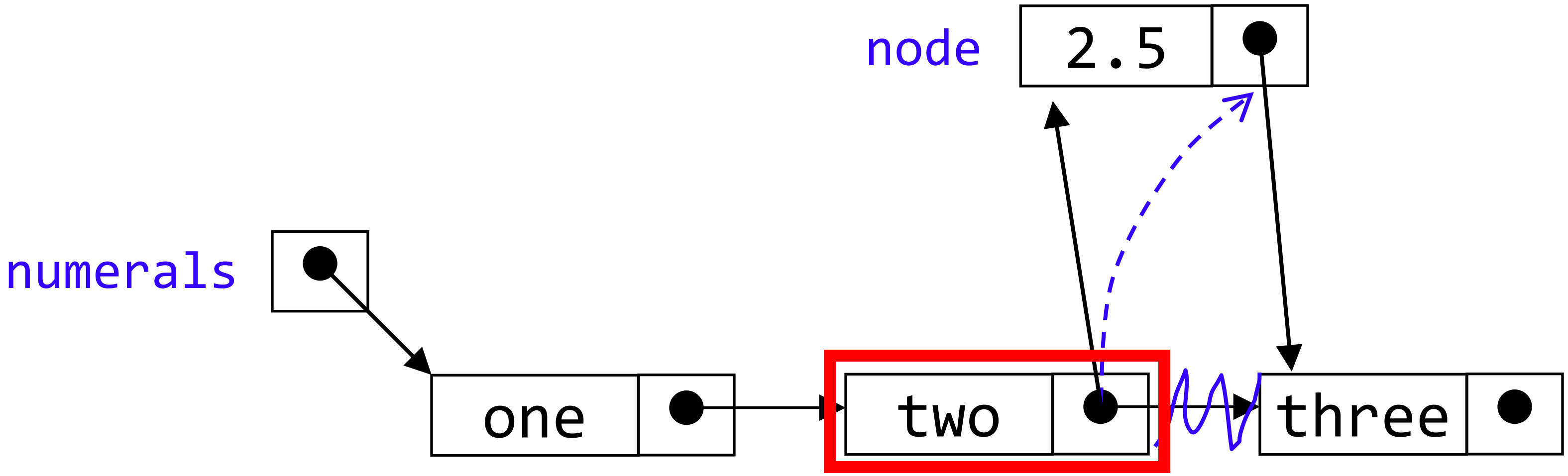
```
void ins_end(value) {
        create a new Node
        store data in new Node
        if (list is Empty)
                set Header to new Node
        else
                set current node to Header
                while current node is not the last node
                        set current node to next node
                current(last) node points to new node
        endif}
}
```

Search or Find element

```
Node find(value) {

        if (list is Empty) return null
        set current node to Header
        while (current node value does not match) AND (current
node is not the last node)
                set current node to next node
        if (current node value matches value)
                return current node
        else
                return null}
```

Inserting after (animation)



Find the node you want to insert after

First, copy the link from the node that's already in the list

Then, change the link in the node that's already in the list

Inserting a node after a given value

```
void ins_after(value) {

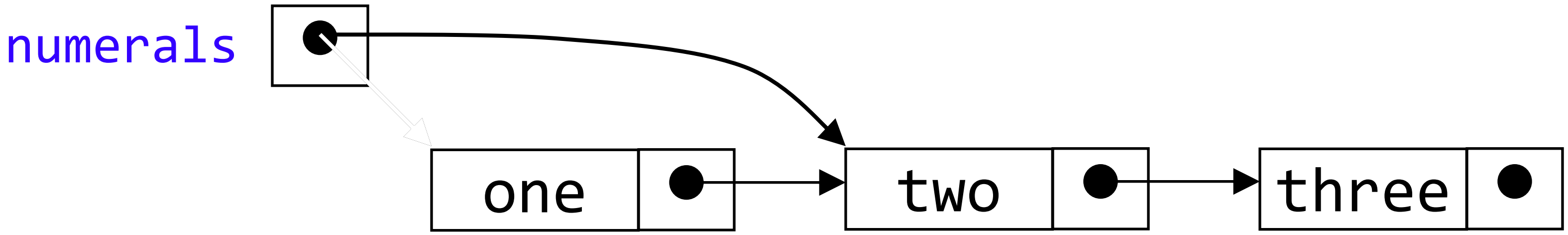
   if (list is Empty) return // cannot insert
   set current node to Header
   while (current node value does not match) AND (current
   node is not the last node)
        set current node to next node
   if (current node value matches value)
        Create new node
        Insert new node
   endif
}
```

Deleting a node from a SLL

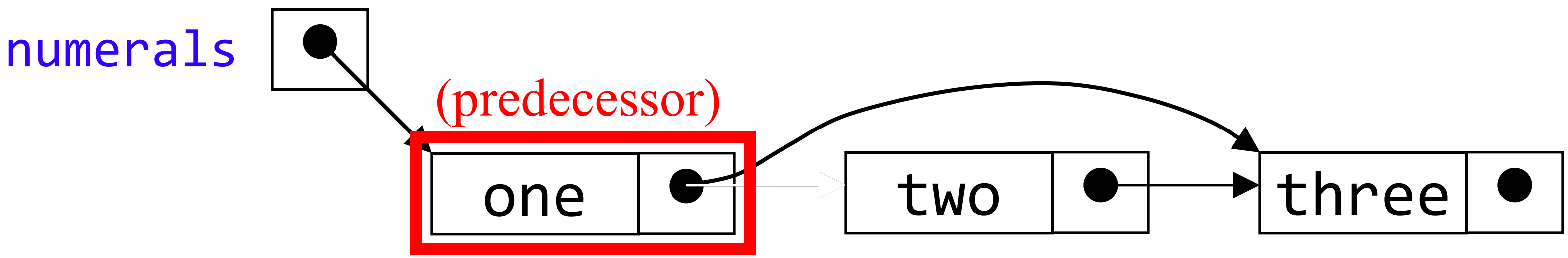
- In order to delete a node from a SLL, you have to change the link in its *predecessor*
- This is slightly tricky, because you can't follow a pointer backwards
- Deleting the first node in a list is a special case, because the node's predecessor is the list header

Deleting an element from a SLL

- To delete the first element, change the link in the header



- To delete some other element, change the link in its predecessor



- Deleted nodes will eventually be garbage collected

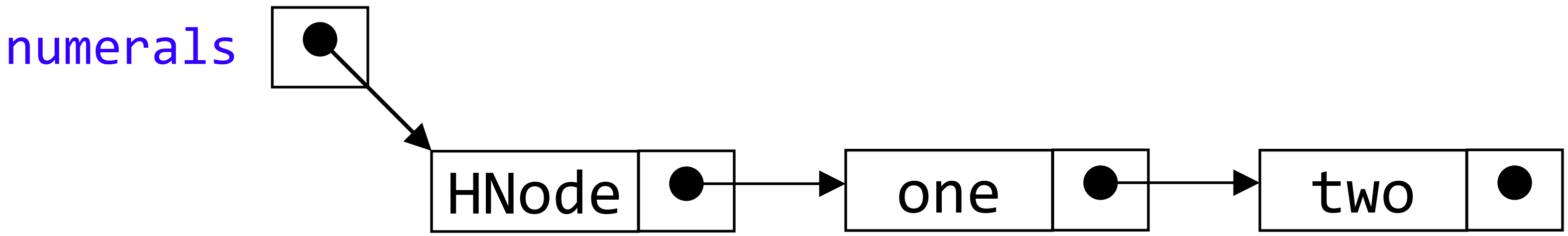
Deleting matched element

```
void del_ele(value) {

        if (list is Empty) return
        if (Header value matches value)
                set Header to next Node
        else
                set current node to Header
                while (current node value does not match) AND (current node is
not the last node)
                        set  previous node to current node
                        set current node to next node
                if (current node value matches value)
                        previous node points to next of current node
        endif
}
```

Using a header node

- A header node is just an initial node that exists at the front of every list, even when the list is empty
- The purpose is to keep the list from being `null`, and to point at the first element



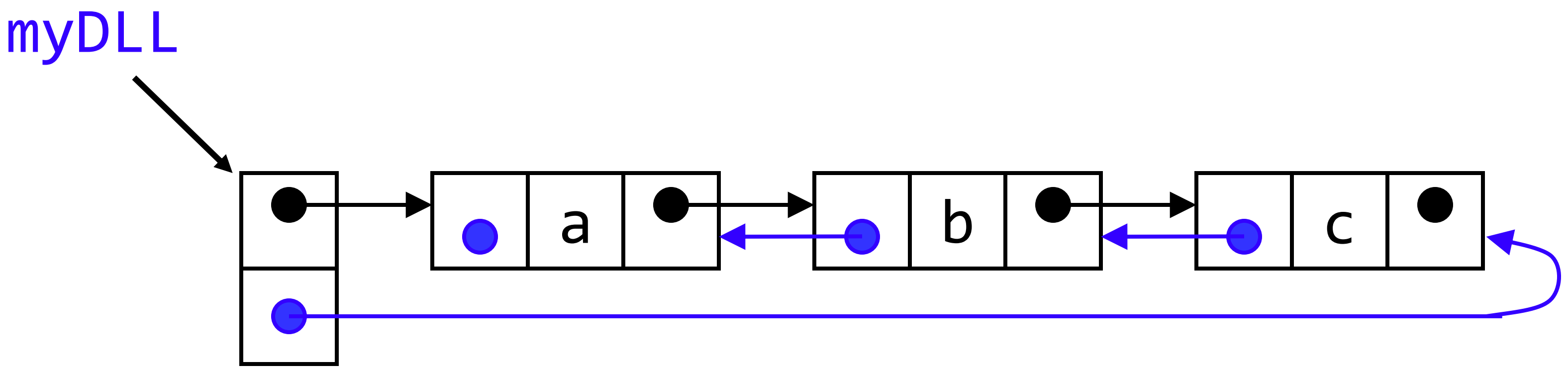
- `Separate Header node is optional`

SLL Problems

- Int IsSorted (struct node * head) return 1 or 0
- OrderInsert (struct node * head)
- RemoveDupes(struct node * head)
- Reverse()
  - Should reverse a SLL using iteration. Use 3 pointers or 2 loops
- Struct node *SortedMerge(struct node *a, struct node *b)
  - a:{1,5,8} b:{2,3,9} should return {1,2,3,5,8,9}
- Struct node *ShuffleMerge(struct node *a, struct node *b)
  - a:{1,3,5} b:{2,4,6} should return {1,2,3,4,5,6}

Doubly-linked lists

- Here is a doubly-linked list (DLL):



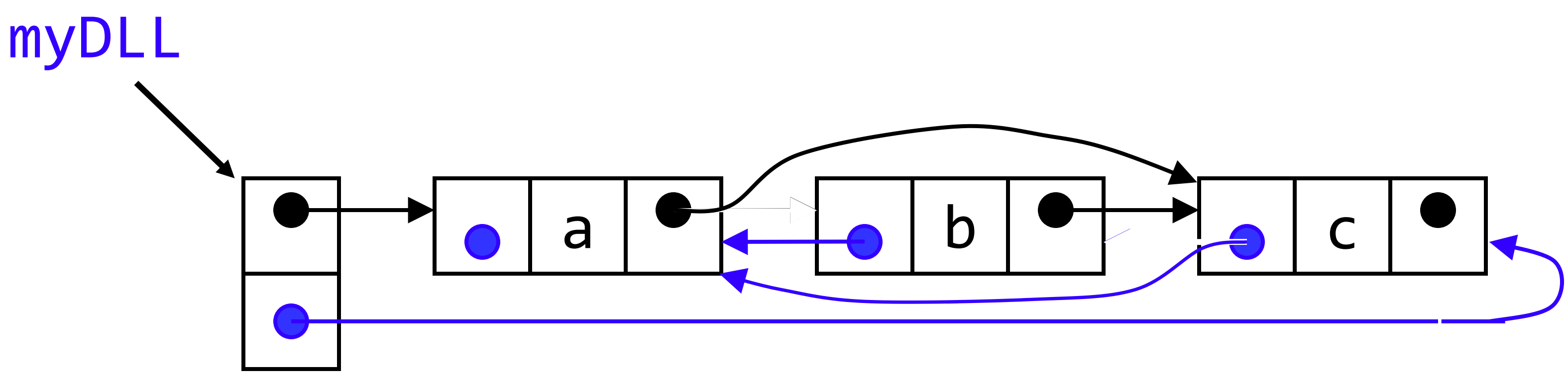
  - Each node contains a value, a link to its successor (if any), *and* a link to its predecessor (if any)
  - The header points to the first node in the list *and* to the last node in the list (or contains null links if the list is empty)

DLLs compared to SLLs

- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

Deleting a node from a DLL

- Node deletion from a DLL involves changing *two* links
- In this example,we will delete node b



- We don't have to do anything about the links in node b
- Garbage collection will take care of deleted nodes
- Deletion of the first node or the last node is a special case

Other operations on linked lists

- Most "algorithms" on linked lists—such as insertion, deletion, and searching—are pretty obvious; you just need to be careful
- Sorting a linked list is just messy, since you can't directly access the n^{th} element—you have to count your way through a lot of other elements

Linked List Versus Array

Guidelines for Choosing between an Array and a Linked List

1. Frequent random access operations → array
2. Operations occur at a cursor → linked list
3. Operations occur at a two-way cursor → DLL
4. Frequent capacity changes → A linked list avoids resizing inefficiency