Data Structures and Algorithms

Lecture 4: Lists, Stacks, and Queues (I)

Lecture outline

- **Lists**, Stacks, Queues
  - Concepts, operations, applications
  - Logical representation of an ADT *versus* Physical implementation of a DS
  - Asymptotic analysis for simple operations
- Dictionaries: concept and usage

Data Structure

- A *construct* that can be defined within a programming language to store a collection of data
  - one may store some data in an array of integers, an array of objects, or an array of arrays

Abstract Data Type (ADT)

- Definition: a collection of *data* together with a set of *operations* on that data
  - specifications indicate *what* ADT operations do, but not *how* to implement them
  - data structures are part of an ADT's implementation
- Programmer can use an ADT without knowing its implementation.

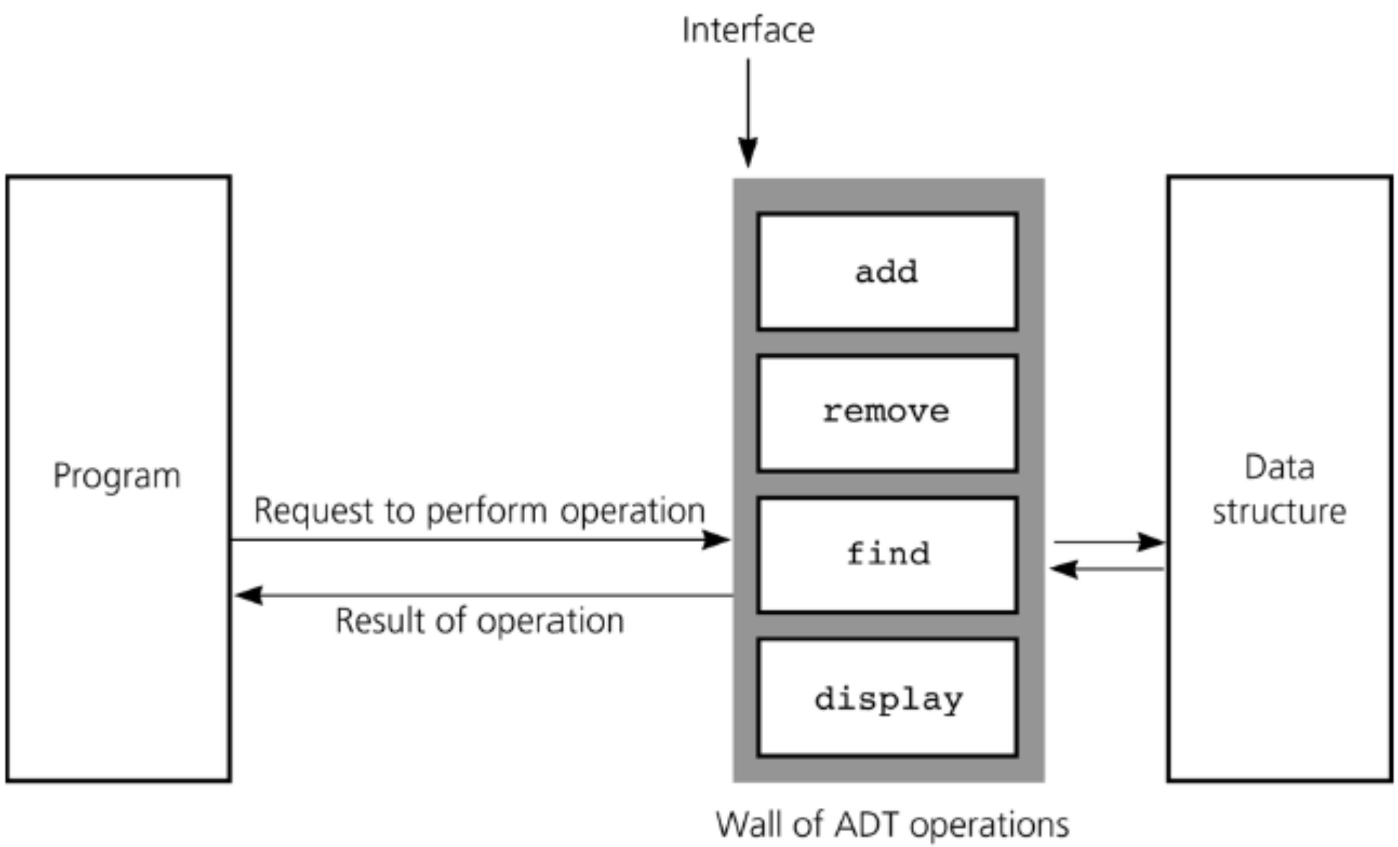
Typical Operations on Data

- Add data to a data collection
- Remove data from a data collection
- Ask questions about the data in a data collection.
  - e.g., what is the value at a particular location, and is x in the collection?

Why ADT

- Hide the unnecessary details
- Help manage software complexity
- Easier software maintenance
- Functionalities are less likely to change
- Localised rather than global changes

Illustration



Lists

Why Lists?

- A *list* is **ALL YOU NEED** to achieve anything promised to a computer
- The rest are all about improving *efficiency*
- Stacks and Queues: list-like structures
  - 1 List, 2 Stacks, 2 Queues are equally capable
- Then why Stacks and Queues?
  - simple, fewer operations
  - come in handy in applications

Lists

- List: a finite sequence of data items
  a1, a2, a3, …, an
- Lists are pervasive in computing
  - e.g. class list, list of chars, list of events
- Typical operations:
  - Creation
  - Insert / remove an element
  - Test for emptiness
  - Find an item/element
  - Current element / next / previous
  - Find k-th element
  - Print the entire list

List feature

- Each list element have its position.
  - Notation: $<a_0, a_1, \ldots, a_{n-1}>$
    - $a_0 = 10$, $a_1 = 9$, $a_2 = 7$, $a_3 = 20$, $a_4 = 8$
- List implementation has a current position.
  - Define the list with **left** and **right** partitions.
    - Either or both partitions may be empty.
  - Partitions are separated by a vertical bar.
    - <20, 23 | 12, 15>

An ADT Interface for List

- Functions
  - `isEmpty`
  - `getLength`
  - `insert`
  - `delete`
  - `Lookup`
  - …
- Data Members
  - `head`
  - `Size`
- Local variables to member functions
  - `cur`
  - `prev`

List ADT: a case

```
template <typename E> class List { // List ADT
public:
   virtual void clear() = 0;
   virtual void insert(const E& item) = 0;
   virtual void append(const E& item) = 0;
   virtual void E remove() = 0;
   virtual void moveToStart() = 0;
   virtual void moveToEnd() = 0;
   virtual void prev() = 0; // move backward
   virtual void next() = 0; // move forward
   virtual int length() const = 0;
   virtual int currPos() const = 0;
   virtual void moveToPos(int pos) = 0;
   virtual const E& getValue() const = 0;
};
```

List ADT Examples

- List: <12 | 32, 15>
  - L.insert(99);
  - Result: <12 | 99, 32, 15>

- Iterate through the whole list:

```
for (L.moveToStart();
  L.currPos()<L.length();
     L.next()) {
  it = L.getValue();
  doSomething(it);
}
```

List Find Function

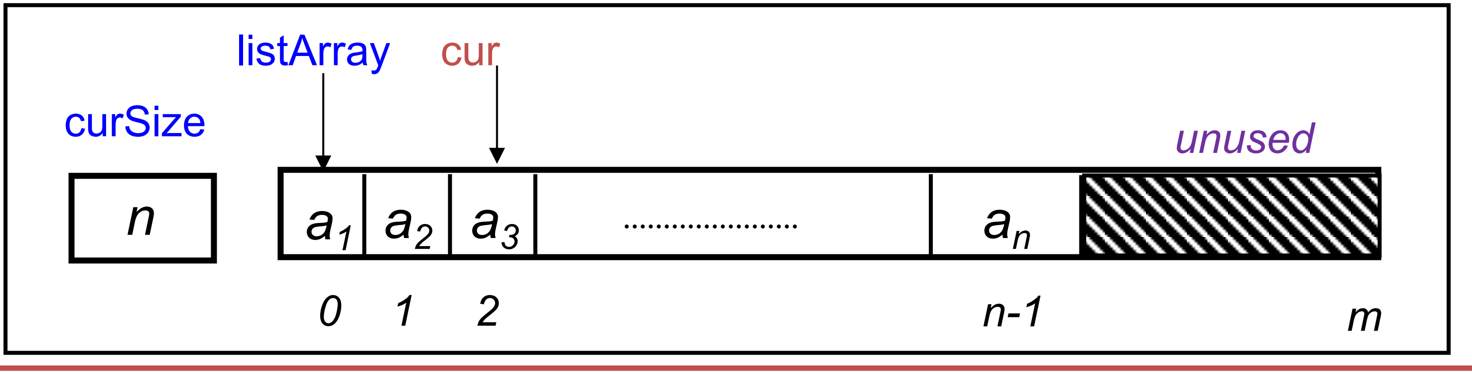
```
/* Return True if 'k' is in list 'L',
    false otherwise */

return_type find(List<int>& L, int k) {
  for (L.moveToStart();
  L.currPos()<L.length(); L.next()) {
    if (k == L.getValue())
       return true; // Found k
  }
  return false;      // k not found
}
```

Two physical implementations

- Array-based lists
- Linked lists

Array-Based List Implementation

- One simple implementation is to use arrays
  - A sequence of *n*-elements
- Maximum size is anticipated a priori.
- Internal variables:
  - Maximum size *maxSize* (m)
  - Current size *curSize* (n)
  - Current index *cur*
  - Array of elements listArray



Array-Based List Class (1)

```
template <typename E>
class Alist : public List<E> {
private:
  E *listArray; // array holding elements
  int maxSize;  // max size of list
  int listSize; // number of list items now
  int curr;     // position of cur. element

public:
  // Constructor
  Alist(int size=10) {
    maxSize = size;
    listSize = curr = 0;
    listArray = new E[maxSize];
  }
```

Array-Based List Class (2)

```
// Destructor
public: ~Alist(){ delete [] listArray; }
public: void clear()
  { listSize = curr = 0; }
```

- **Move position functions**

```
public:
  void moveToStart() { curr = 0; }
  void moveToEnd() { curr = listSize; }
  void prev() { if (curr != 0) curr--; }
  void next()
    { if (curr < listSize) curr++; }
  int length() { return listSize; }
  int currPos() { return curr; }
```

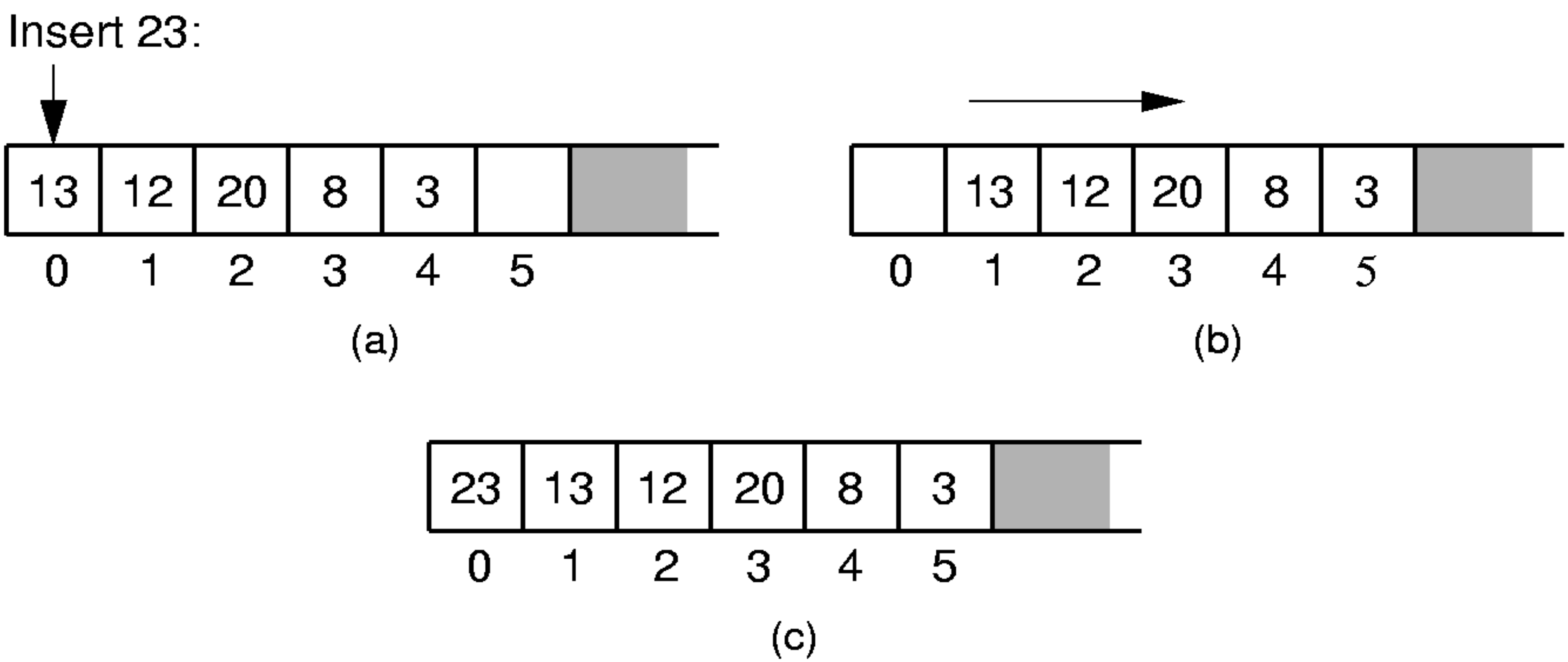
Array-Based List Class (3)

```
// Set current list position to 'pos'
public: void moveToPos(int pos) {
   if( pos < 0 || pos >= listSize){
      cout << "Position out of range" <<endl;
      abort();
   }
   curr = pos;
}

// Return current element
public: E& getValue() const {
   assert(curr >= 0 && curr < listSize);

   return listArray[curr];
}
```

Insert an element

- An insert operation at position 0

<|13, 12, 20, 8, 3, ...>



Insert

```
/** Insert "it" at current position */

public: void insert(E it) {
  // List capacity exceeded
  assert(listSize < maxSize );
  for (int i=listSize; i>curr; i--)
    listArray[i] = listArray[i-1];
  listArray[curr] = it;
  listSize++;
}
```

Append

```
/** Append "it" at the end of the list */

public: void append(E it) {
  // List capacity exceeded
  assert(listSize < maxSize);

  listArray[listSize] = it;
  listSize++;
}
```

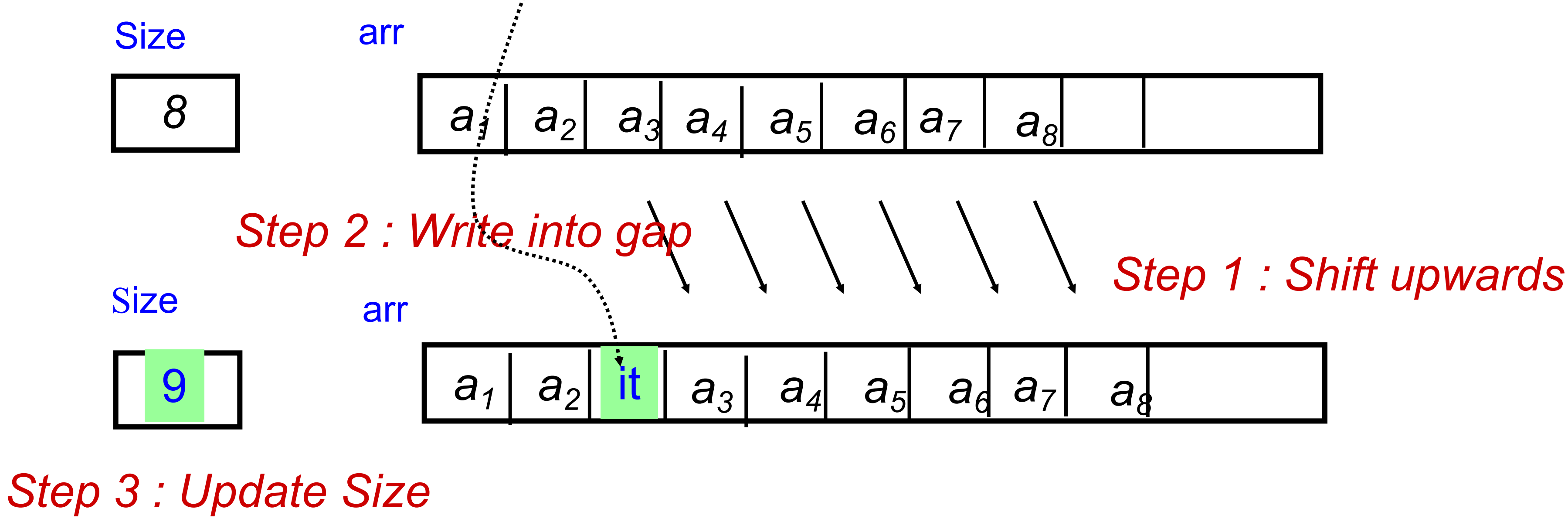
Remove

```
/** Remove and return the current element */

public: E remove() {
  if ( curr < 0 || curr >= listSize)
    return NULL;
  E it = listArray[curr];
  for(int i=curr; i<=listSize-2; i++)
    listArray[i] = listArray[i+1];

  listSize--;
  return it;
}
```

Inserting Into an Array

- While retrieval is very fast, insertion and deletion are very slow
  - Insert has to shift upwards to create gap

Example : insert(2, it, arr)



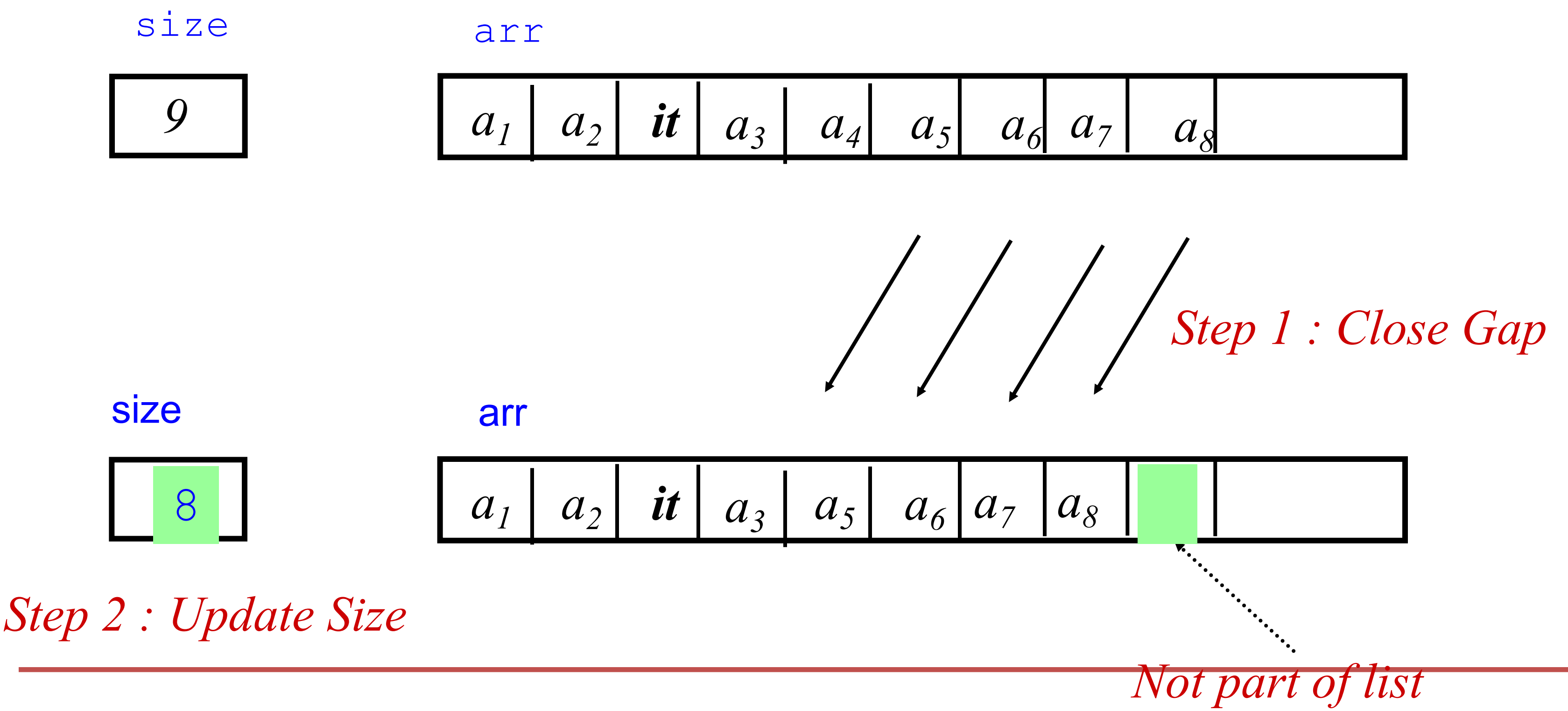
Coding

```
struct array_list {
   int arr[MAX];
   int max;
   int size;
} LIST;


void insert(int j, int it, LIST *pl)
   { // pre : 1<=j<=size+1

      int i;

      for (i=pl->size; i>=j; i=i-1)
                                          // Step 1: Create gap
         { pl->arr[i+1]= pl->arr[i]; };

      pl->arr[j]= it;// Step 2: Write to gap

      pl->size = pl->size + 1; // Step 3: Update size
   }
```

Deleting from an Array

- Delete has to shift downwards to close gap of deleted item

Example: `deleteItem(4, arr)`



Coding

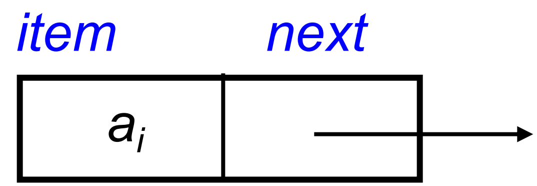
```
void delete(int j, LIST *pl)
{ // pre : 1<=j<=size
  for (i=j+1; i<=pl->size; i=i+1)
   // Step1: Close gap
    { pl->arr[i-i]=pl->arr[i]; };
   // Step 2: Update size
    pl->size = pl->size - 1;
  }
```

Two physical implementations

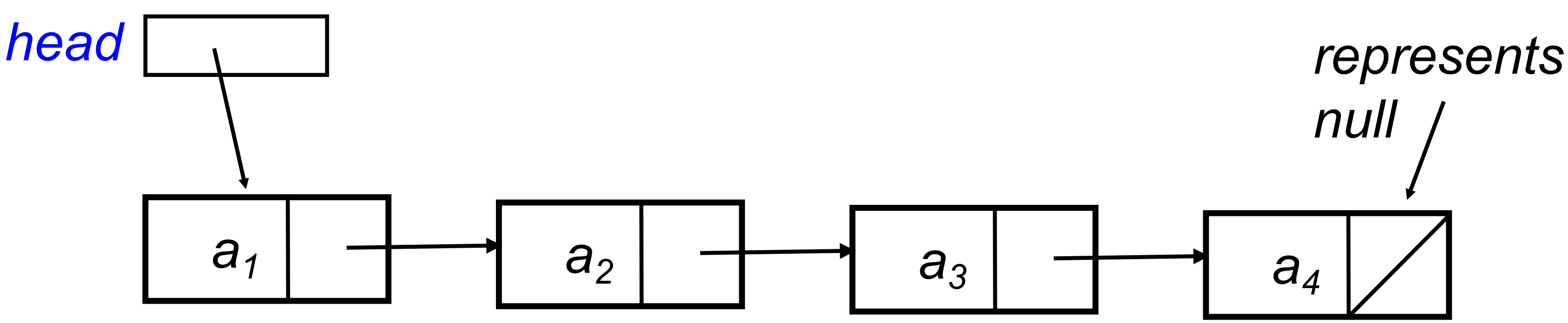
- Array-based lists
- Linked lists

Linked List Approach

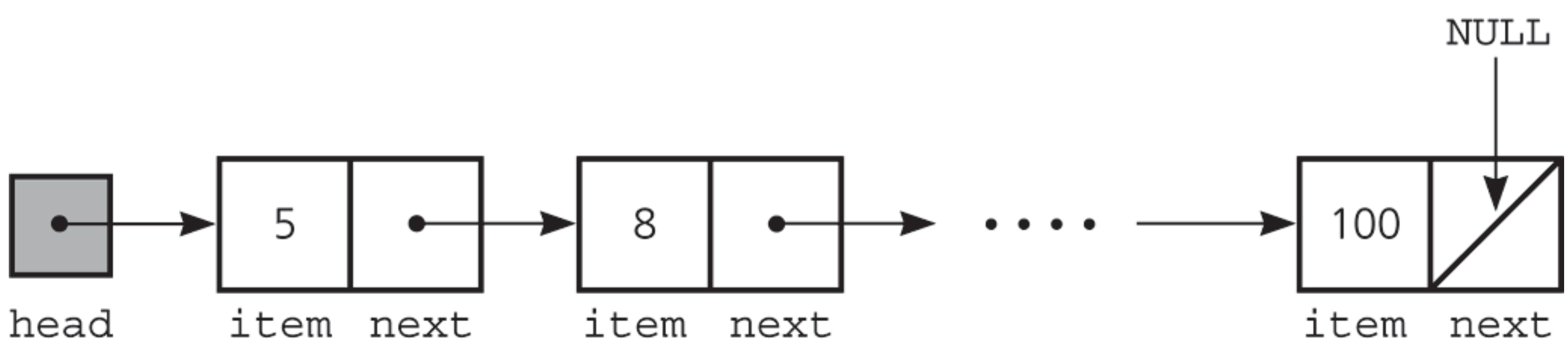
- Main problem of array is the slow deletion/insertion since it has to shift items in its *contiguous* memory
- **Solution**: linked list where items need *not be contiguous* with nodes of the form



- Sequence (list) of four items $< a_1, a_2, a_3, a_4 >$ can be represented by:



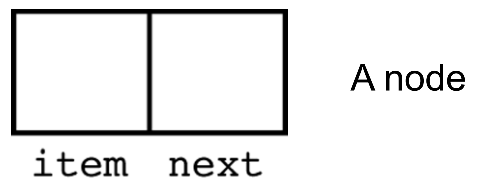
A Sample Linked List



Pointer-Based Linked Lists

- A node in a linked list is usually a `struct`

```
struct Node
{ int item
  Node *next;
}; //end struct
```



- A node is dynamically allocated

```
Node *p;
p = malloc(sizeof(Node));
```

Pointer-Based Linked Lists

- The head pointer points to the first node in a linked list
- If head is *NULL*, the linked list is empty
  - `head=NULL`
- `head=malloc(sizeof(Node))`

Linked List Node Class

```
// Singly linked list node
template <typename E> class Link {
public:
  E element;
  Link *next;
  // Constructors
  Link(const E* elemval, Link* nextval = NULL)
    {element = elemval; next = nextval;}
  Link(Link* nextval = NULL) {
    next = nextval;
  }
}
```

Linked List Class (1)

```
template <typename E>
class LList : public List<E> {
private:
  Link<E>* head; // pointer to list header
  Link<E>* tail; // pointer to last element
  Link<E>* curr; // access to current element
  int cnt;  // size of list

public:
  //Constructor
  LList() {
    curr = tail = head = new Link<E>(NULL);
    cnt = 0;
  }
```

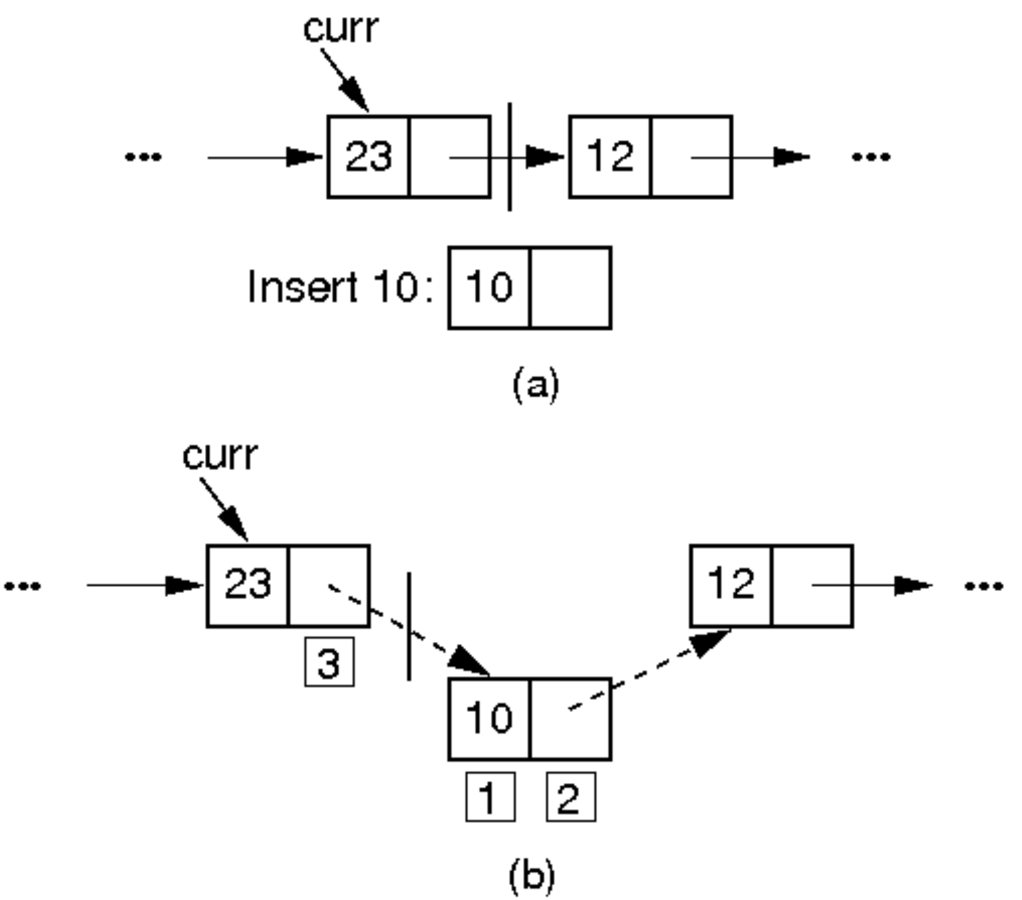
Linked List Class (2)

```
public: void clear() {
  curr= head->next; //keep the head node
  Link<E>* tmp;
  while( curr != NULL){
      tmp = curr;
      curr = curr->next;
      delete tmp;
  }
  head->next = NULL;
  curr = tail = head;
  cnt = 0;
}

~LList(){
   clear();
   delete head;
}
```

Linked List Class (3)

```
public:
  void moveToStart() { curr = head; }
  void moveToEnd() { curr = tail; }
  int length() { return cnt; }
  void next() {
    if (curr != tail) { curr = curr->next; }
  }

  const E& getValue() const {
    // Nothing to get;
    assert(curr->next != NULL);

    return curr->next->element;
  }
```

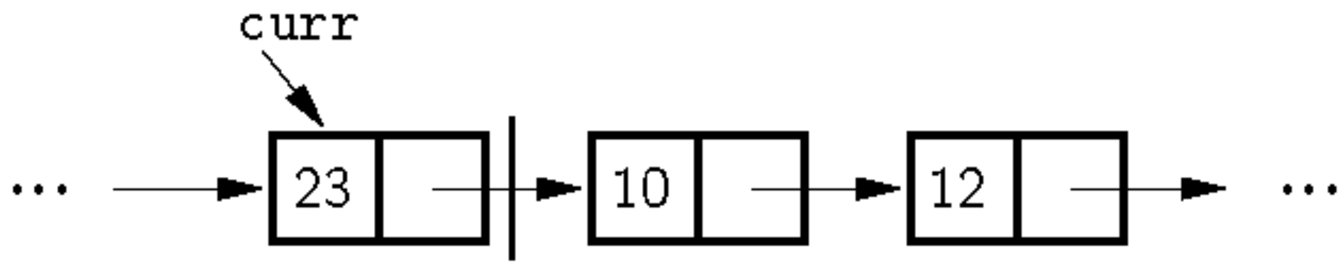
Insertion



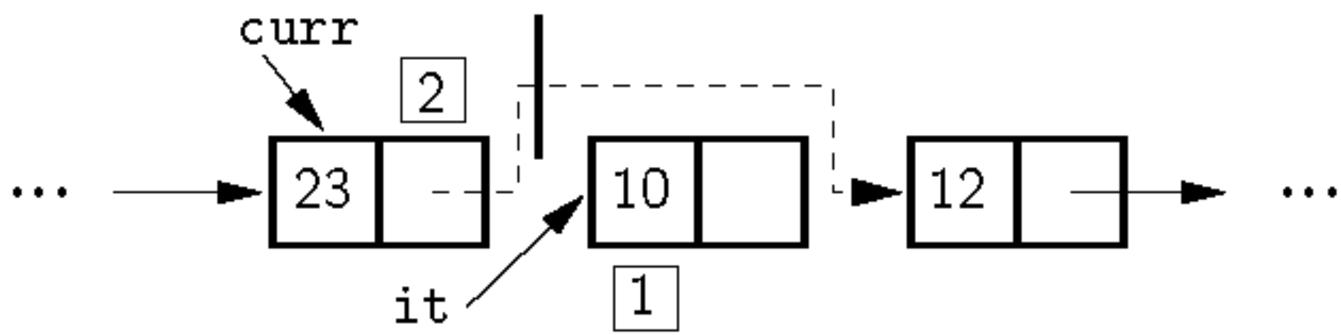
Code case of Insert/Append

```
// Insert "it" at current position
void insert(E& it) {
  Link<E>* tmp = new Link<E>(it, curr->next);
  curr->next = tmp;

  if (tail == curr) tail = curr->next;
  cnt++;
}

// Append "it" to list
void append(E& it) {
  tail->next = new Link<E>(it, NULL);
  tail = tail->next;
  cnt++;
}
```

Removal



(a)



(b)

Code case of remove

```
/** Remove and return current element */
E remove() {
  // if no elements;
  assert(curr->next != NULL);

  if (tail == curr->next) tail = curr;

  // tmp points to the node to be deleted
  Link<E>* tmp = curr->next;
  E it = tmp->element;

  curr->next = tmp->next;
  delete tmp;
  cnt--;

  return it;
}
```

Previous

```
/** Move curr one step left;
    no change if already at front */

void prev() {
  if (curr == head) return;

  Link<E>* tmp = head;
  // March down list until previous found
  while (tmp->next != curr)
    tmp = tmp->next;
  curr = tmp;
}
```

Get/Set Position

```
/** Return position of the current element */
int currPos() {
  Link<E>* tmp = head;
  int i;
  for (i=0; tmp != curr; i++)
    tmp = tmp->next;
  return i;
}

/** Move down list to "pos" position */
void moveToPos(int pos) {
  // if position is out of range;
  assert( pos>=0 && pos<cnt);

  curr = head;
  for(int i=0; i<pos; i++)
    curr = curr->next;
}
```

Traverse a Linked List

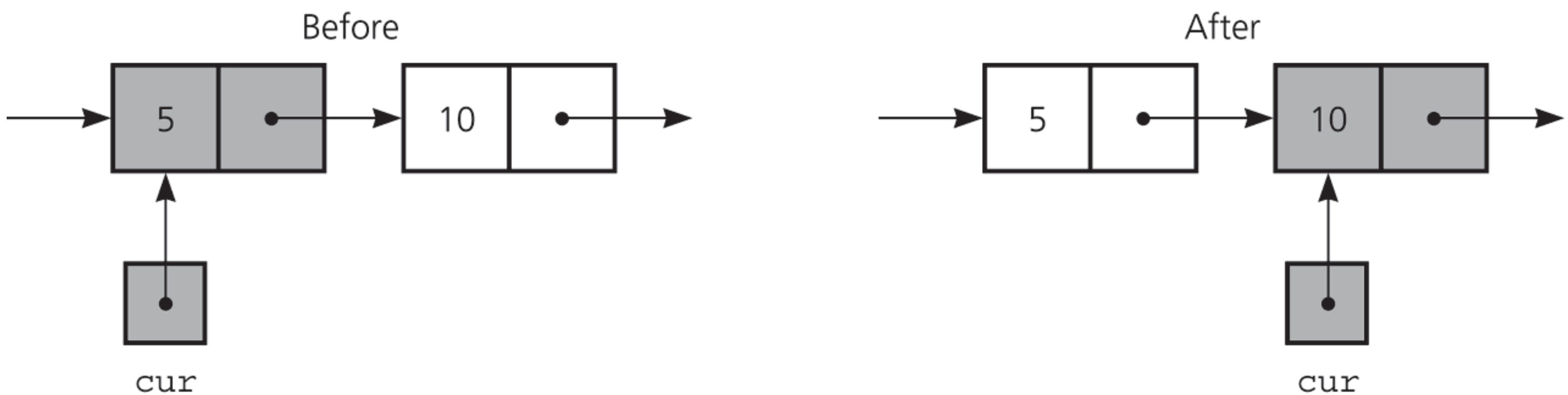
- Reference a node member with the -> operator

```
p->item;
```

- A traverse operation visits each node in the linked list
  - A pointer variable cur keeps track of the current node

```
for (Node *cur = head;
     cur != NULL; cur = cur->next)
    x = cur->item;
```

Traverse a Linked List



The effect of the assignment `cur = cur->next`

Delete a Node from a Linked List

- Deleting an interior/last node

```
prev->next=cur->next;
```

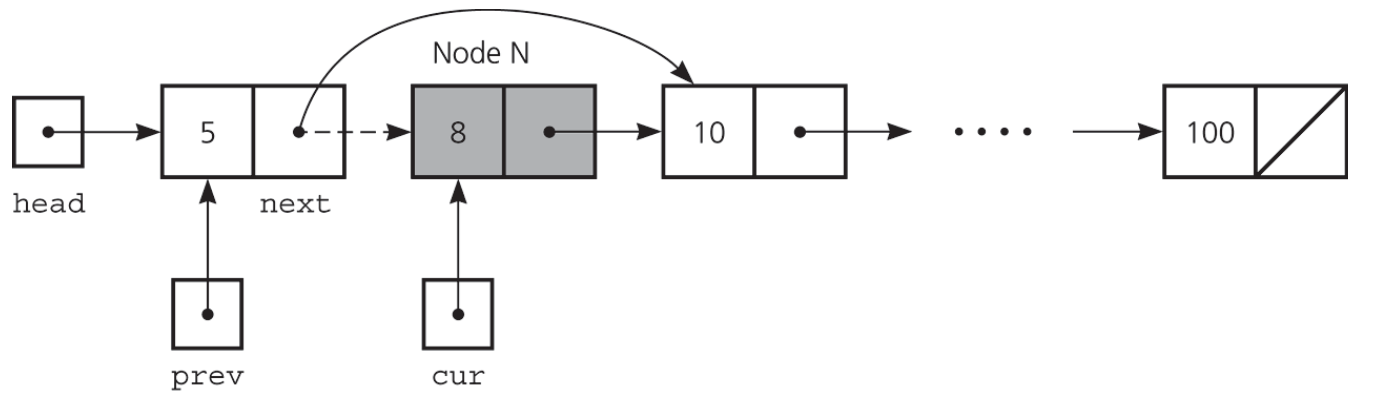
- Deleting the first node

```
head=head->next;
```

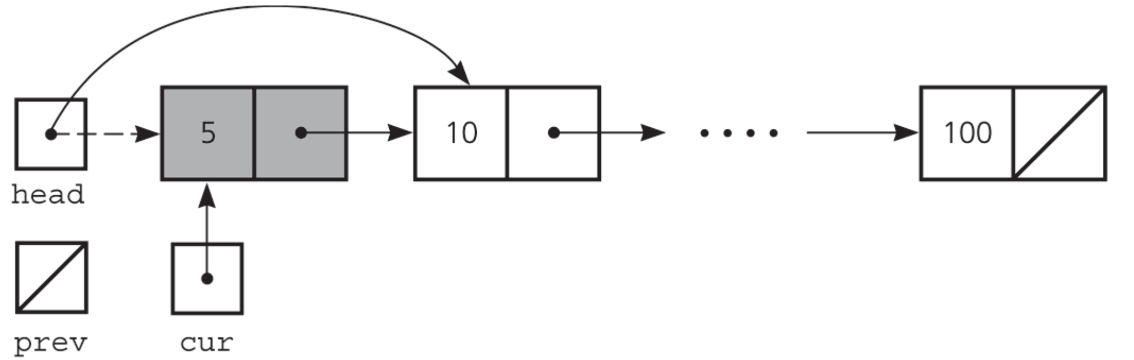
- Return deleted node to system

```
cur->next = NULL;
free(cur);
cur=NULL;
```

Delete a Node from a Linked List



Deleting a node from a linked list

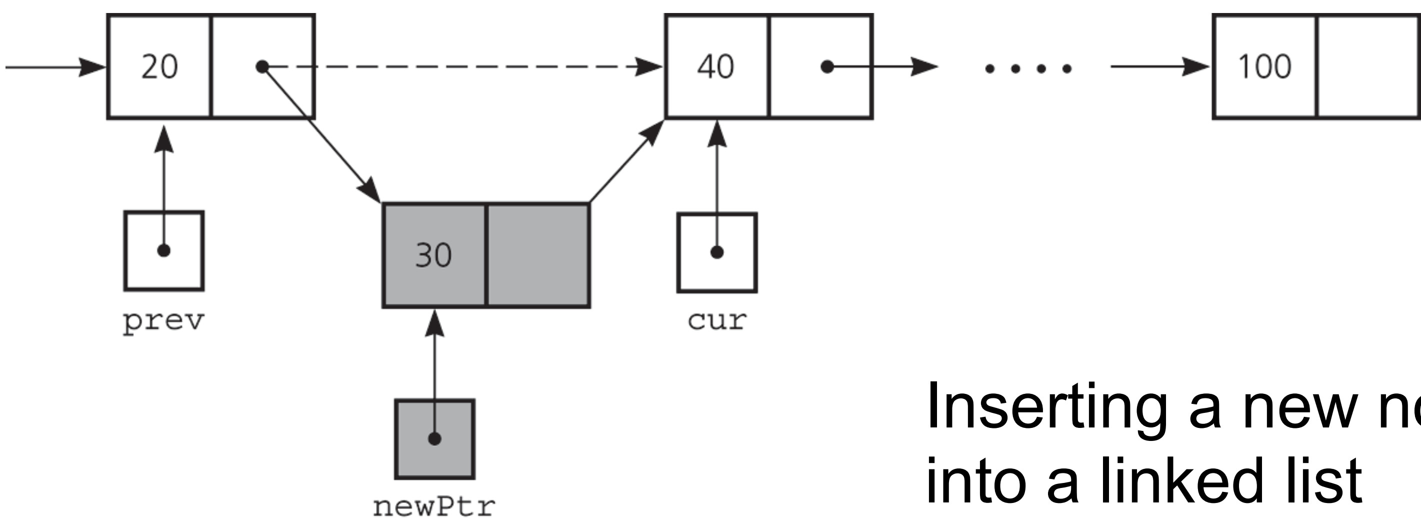


Deleting the first node

Insert a Node into a Linked List

- To insert a node between two nodes

```
newPtr->next = cur;
prev->next = newPtr;
```

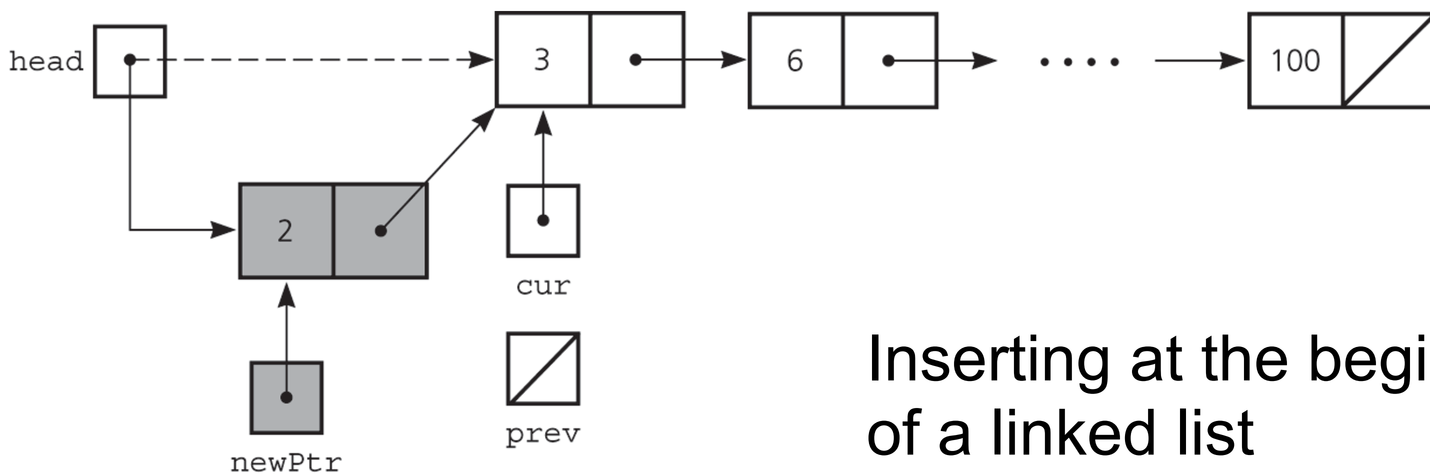


Inserting a new node into a linked list

Insert a Node into a Linked List

- To insert a node at the beginning of a linked list

```
newPtr->next = head;
head = newPtr;
```

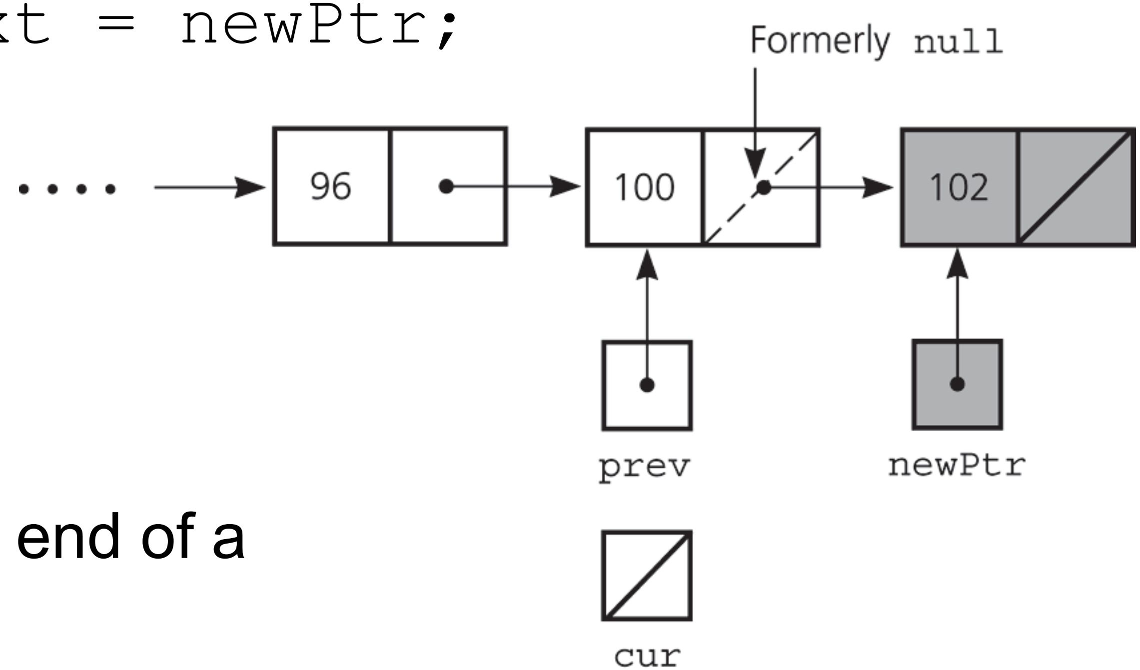


Inserting at the beginning of a linked list

Insert a Node into a Linked List

- Inserting at the end of a linked list is not a special case if `cur` is *NULL*

```
newPtr->next = cur;
prev->next = newPtr;
```



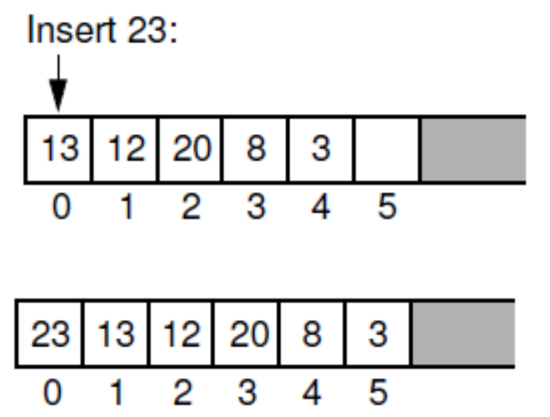
Inserting at the end of a linked list

Look up

```
BOOLEAN lookup (int x, Node *L)
 {  if (L == NULL)
       return FALSE
    else if (x == L->item)
        return TRUE
    else
        return lookup(x, L-next);
 }
```

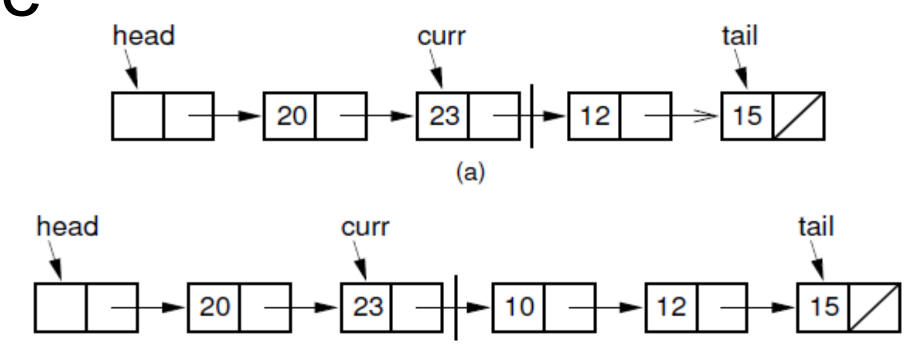
Array-based lists *versus* linked list

- The memory addresses of the elements in an array list are *in increasing order*
  - Assume that the start address of the array is 1,000
  - The addresses of elements 13, 12, 20, 8, 3 are 1,000, 1,004, 1,008, 1,012, and 1,016, respectively
- The addresses of the elements after current position increases by 4 with an insertion, if an int varaible takes 4 bytes memory



Array-based lists *vs* linked list (cont.)

- The memory addresses of the elements in a linked list have no relationship with their positions in the list
  - Allocated by the operating system
    - e.g., the memory addresses of 20, 23, 12, 15 are 1,000, 940, 1076, 40
- The addresses of the elements already in the list will not change after an insertion



Comparison of Implementations

Array-Based Lists:

- Insertion and deletion are $\Theta(n)$.
- Prev and direct access are $\Theta(1)$.
- Array must be allocated in advance.
- No overhead if all array positions are full.

Linked Lists:

- Insertion and deletion are $\Theta(1)$.
- Prev and direct access are $\Theta(n)$.
- Space grows with number of elements.
- Every element requires overhead.

Space Comparison

"Break-even" point:

$$DE = n(P + E);$$

$$n = \frac{DE}{P + E}$$

E: Space for data value.
P: Space for pointer.
n: number of elements in the list
D: Number of elements in array with $D >= n$

Freelist

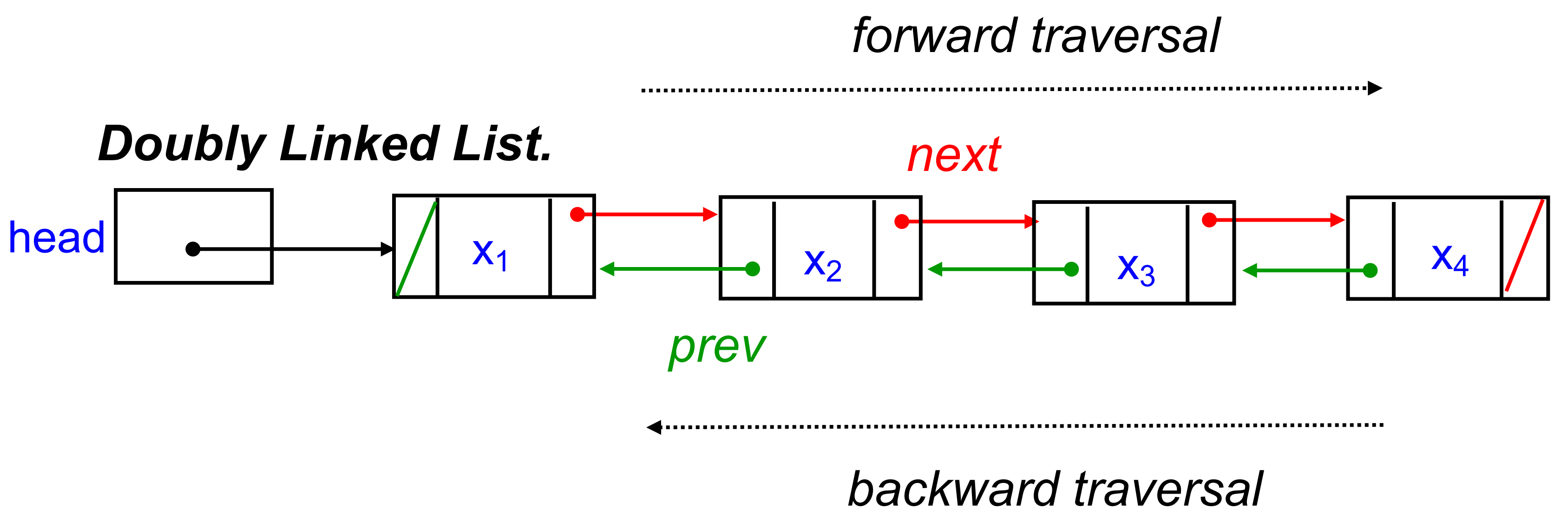
- System new and delete are slow.
- Consider there are many interwoven insert and remove operations

  list.insert(10), list.remove(); list.remove();…,
  list.insert(20),…

- Solution
  - keep the nodes removed in a free list by yourself, and do not call the system delete
  - Allocate a new node from the free list first if there are some; otherwise, call the system new
  - Delete all nodes in the free list when no needing
- See the textbook for details

Doubly Liked Lists

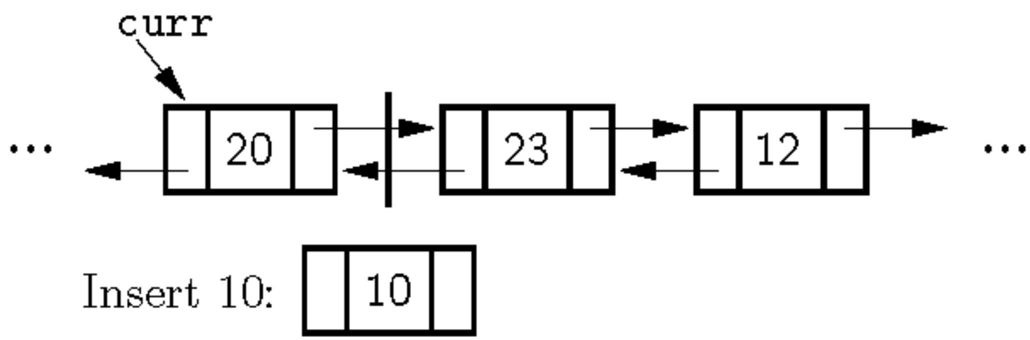
- Frequently, we need to traverse a sequence in BOTH directions efficiently
- *Solution* : Use doubly-linked list where each node has two pointers



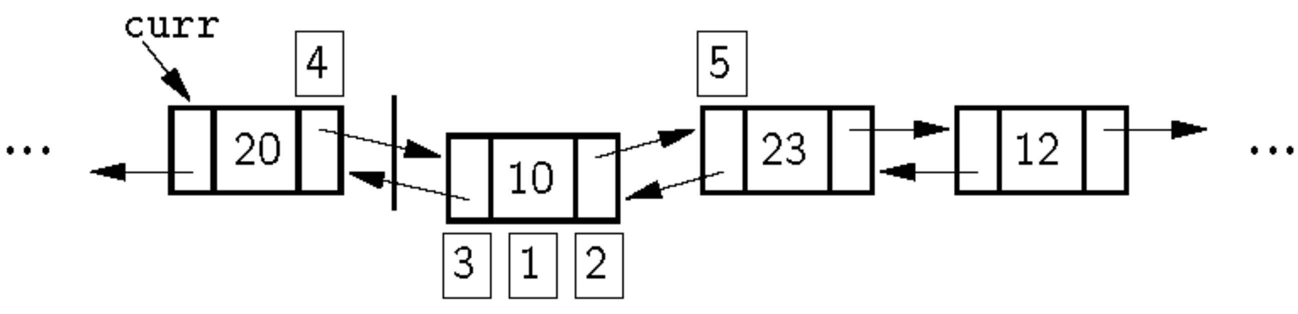
Doubly linked list node

```
template <typename E> class DLink{
public:
    E element;
    DLink* next;
    DLink* prev;

 //Constructors
  DLink(const E& it, DLink* p, DLink* n){
      element = it;
      prev = p;  next = n;
  }
 DLink(DLink* p=NULL, DLink* n=NULL){
      prev = p;
      next = n;
  }
};
```

Doubly Linked Insert



(a)

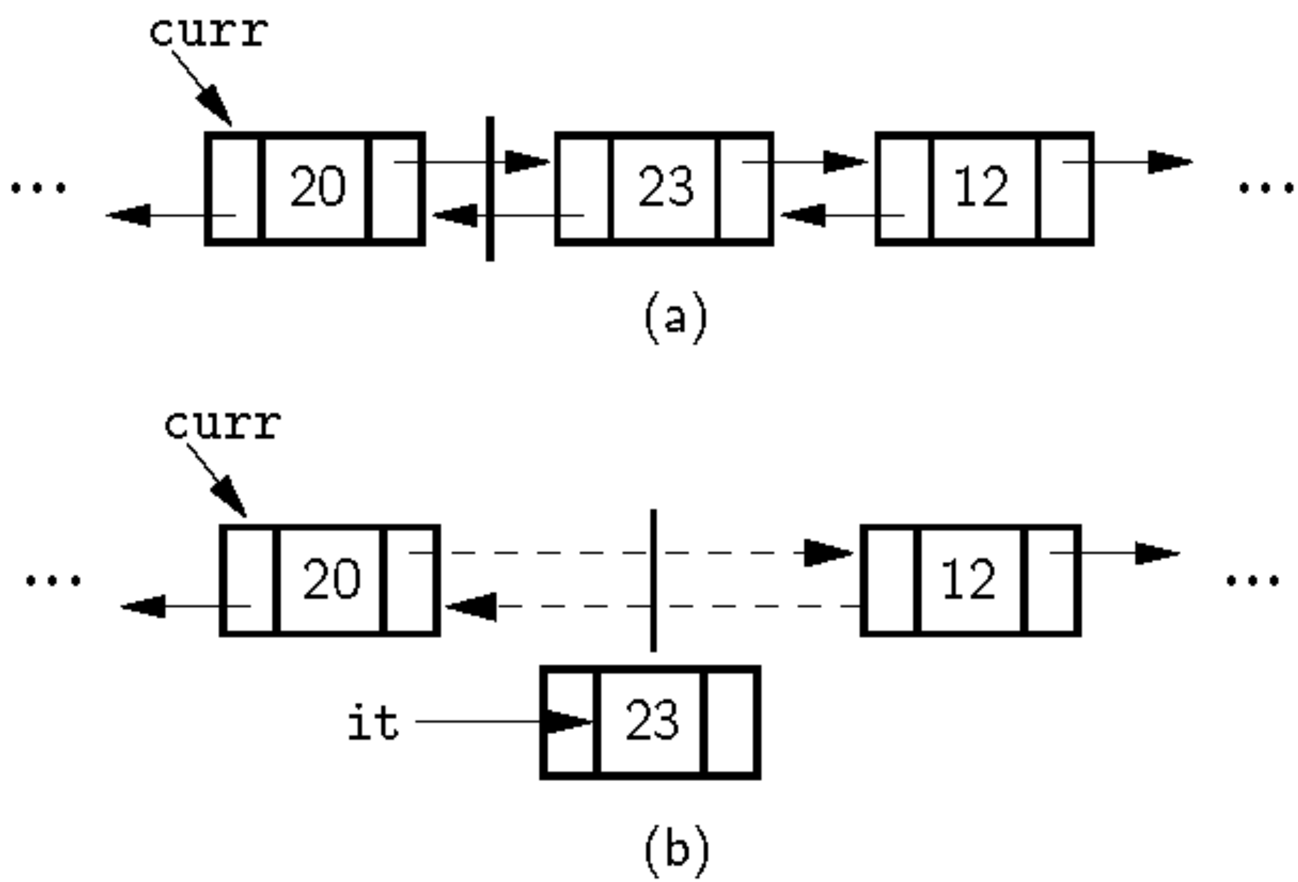


(b)

Doubly Linked Insert

```
// Insert "it" at current position

void insert(E it) {
  DLink<E> *tmp = new DLink<E>(it, curr,
  curr->next );

  curr->next = tmp;

  DLink<E> *pNext = tmp->next;
  pNext->prev= tmp;

   cnt++;
}
```

Doubly Linked Remove



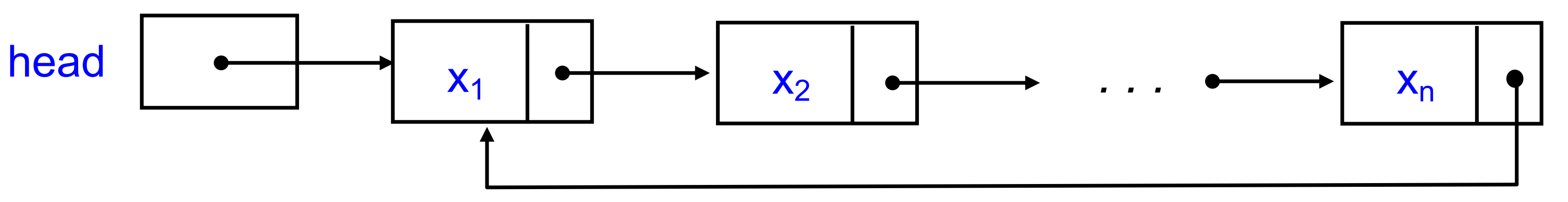
Doubly Linked Remove

```
// Remove and return current element
E remove() {
  if (curr->next == tail) return NULL;

  DLink<E> *tmp = curr->next;
  E it = tmp->element;

  curr->next = tmp->next;
  (tmp->next)->prev = curr;

  cnt--;
  delete tmp;
  return it;
}
```

Circular Linked Lists

- May need to cycle through a list repeatedly, e.g. round robin system for a shared resource
- *Solution* : Have the last node point to the first node

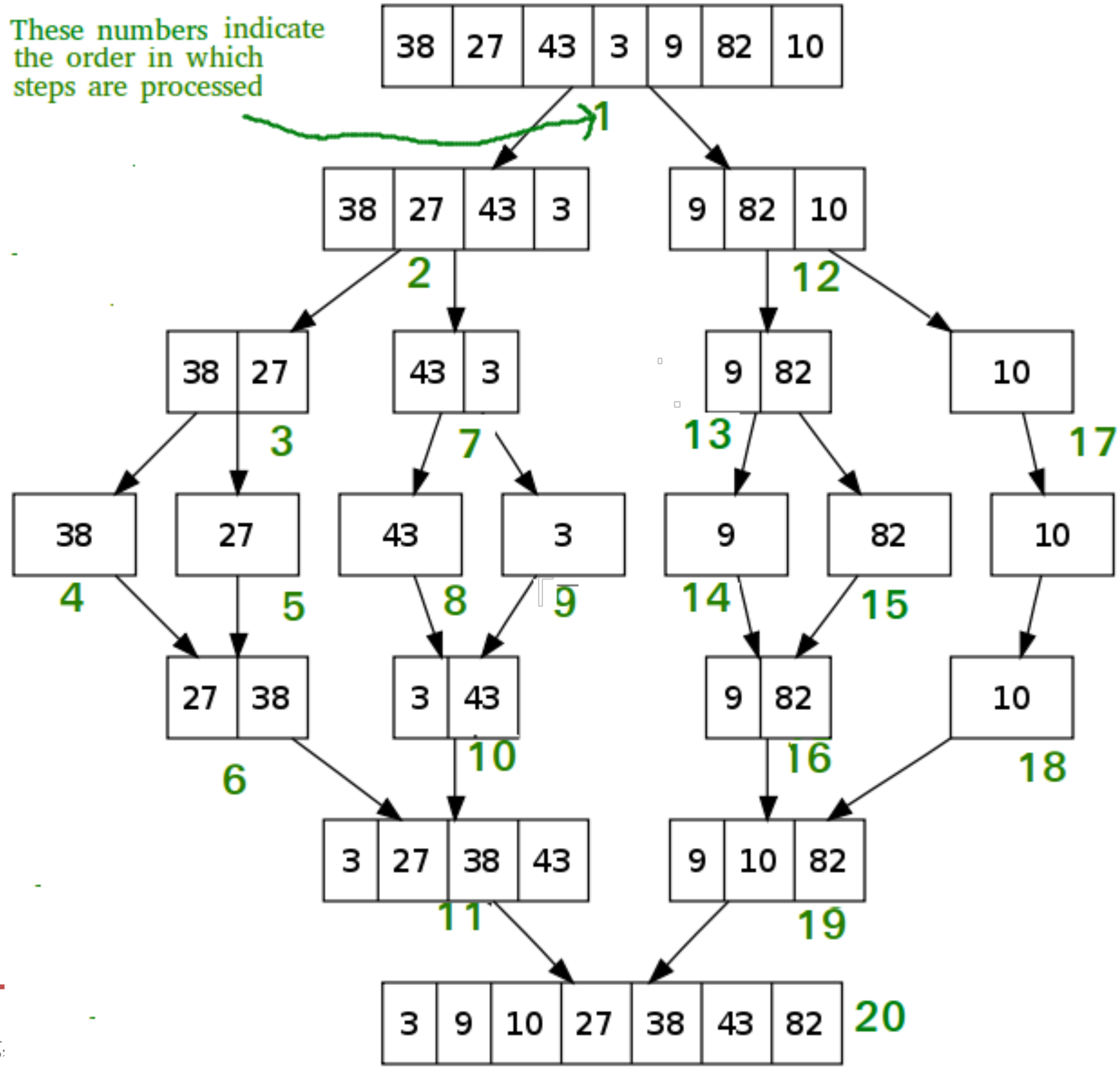
Circular Linked List.



An application of lists -- merge sort

38	27	43	3	9	82	10

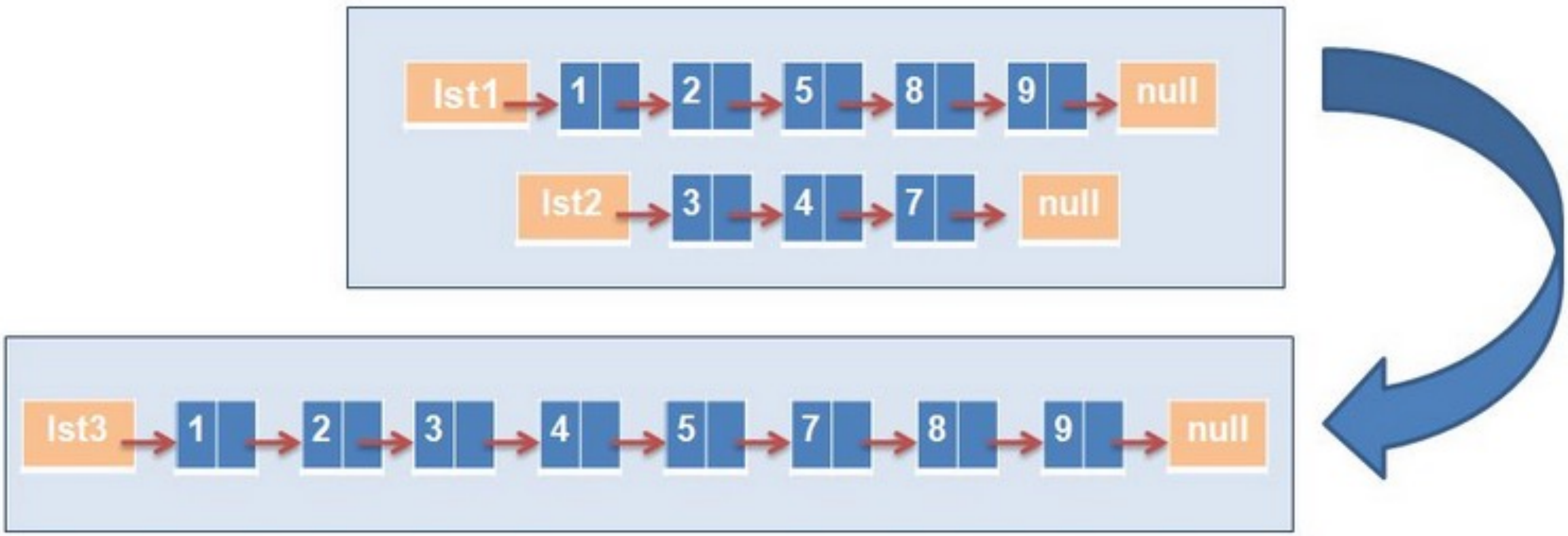
An application of lists -- merge sort



Merge Sort

1. If there is only one number in the list, return;
2. **Split** a list into two sub-lists with almost equal length
3. **Recursively sort** the two sub-lists, where the numbers in each sub-lists are in increasing order
4. **Merge** the two sub-lists into one list such that the number the merged list are in increasing order

How to merge two sorted linked-lists?



Summary

- Array-based lists
  - Fast random access
  - Insertion and removal take long time
- Linked lists
  - Slow for random access
  - Fast insertion and removal
- Singled and doubly linked list
  - The notion of curr
  - Add head and/or tail nodes for convenient coding
  - Pay attention to special cases