The Second Course

List

Lists

- A *list* is a finite, ordered sequence of data items called *elements*.
- Each list element has a data type.
- The *empty* list contains no elements.
- The *length* of the list is the number of elements currently stored.
- The beginning of the list is called the *head*,the end of the list is called the *tail*.

Lists

- *Sorted lists* have their elements positioned in ascending order of value
- *unsorted lists* have no necessary relationship between element values and positions.
- Notation: $(a_0, a_1, \ldots, a_{n-1})$
- What operations should we implement?

Operations

- *Construct* the list, leaving it empty
- Determine whether the list is *empty* or not
- Determine whether the list is *full* or not
- Find the *size* of the list
- *Clear* the list to make it empty
- *Insert* an entry at a specified position of the list

Operations

- *Remove* an entry from a specified position in the list
- *Retrieve* the entry from a specified position in the list
- *Replace* the entry at a specified position in the list
- *Traverse* the list, performing a given operation on each entry

Position Number in a List

- To find an entry in a list, we use an integer that gives its position within the list.

- We shall number the positions in a list so that the first entry in the list has position 0, the second position 1, and so on.

Position Number in a List

- Locating an entry of a list by its position is superficially like indexing an array, but there are important differences.
- If we *insert* an entry at a particular position, then the position numbers of all later entries increase by 1.
- If we *remove* an entry, then the positions of all following entries decrease by 1.

Position Number in a List

- The position number for a list is defined *without* regard to the implementation.
- For a contiguous list, implemented in an array, the position will indeed be the index of the entry within the array.
- But we will also use the position to find an entry within linked implementations of a list, where no indices or arrays are used at all.

List ADT(1)

```
class List {                                // List class ADT
  public:
    List(int =LIST_SIZE);                   // Constructor
    ~List();                                // Destructor
    void clear();                           // Remove all
    void insert(const Elem);                // Insert Elem at curr
    void append(const Elem);                // Insert Elem at tail
    Elem remove();                          // Remove and return Elem
    void setFirst();                        // Set curr to first pos
```

Lists ADT(1)

```
  void prev();                        // Move curr to prev pos
  void next();                        // Move curr to next pos
  int length() const;                 // Return current length
  void setPos(int);                   // Set curr to position
  void setValue(const Elem);          // Set current value
  Elem currValue() const;             // Return current value
  bool isEmpty() const;               // TRUE if list is empty
  bool isInList() const;              // TRUE if curr in list
  bool find(int);                     // Find value
};
```

List ADT(2)

- List::List();
  - The List has been created and is initialized to be empty
- void List::clear();
  - All List entries have been removed; the List is empty
- int List::size() const;
  - The function returns the number of entries in the List

List ADT(2)

- bool List ::empty( ) const;
  - The function returns true or false according to whether the List is empty or not.
- bool List::full() const;
  - The function returns true or false according to whether the List is full or not

List ADT(2)

- Error_code List::insert(int position,
  const List_entry &x);
  - If the List is not full and $0 \le position \le n$, where n is the number of entries in the List, the function succeeds: Any entry formerly at position and all later entries have their position numbers increased by 1, and x is inserted at position in the List

List ADT(2)

- Error_code List::remove(int position,

  List_entry &x);

  - If $0 \le$ position $<$ n, where n is the number of entries in the List, the function succeeds: The entry at position is removed from the List, and all later entries have their position numbers decreased by 1. The parameter x records a copy of the entry formerly at position

List ADT(2)

- Error_code List::retrieve(int position,
  List_entry &x) const;
  - If 0 ≤ position< n, where n is the number of entries in the List, the function succeeds: The entry at position is copied to x; all List entries remain unchanged
  - Else: The function fails with a diagnostic error code

List ADT(2)

- Error_code List::replace(int position,
  const List_entry &x);
  - If 0 ≤ position< n, where n is the number of entries in the List, the function succeeds: The entry at position is replaced by x; all other entries remain unchanged
  - Else: The function fails with a diagnostic error code

List ADT(2)

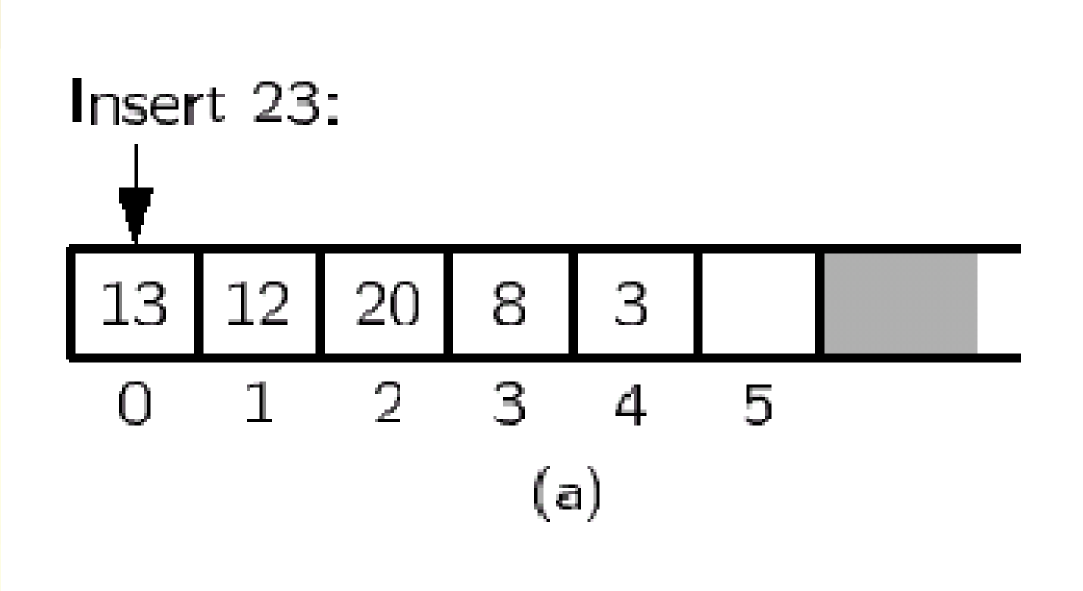
- void List::traverse(void (*visit)(List_entry &));
  - The action specified by function *visit has been performed on every entry of the List, beginning at position 0 and doing each in turn

List ADT Examples

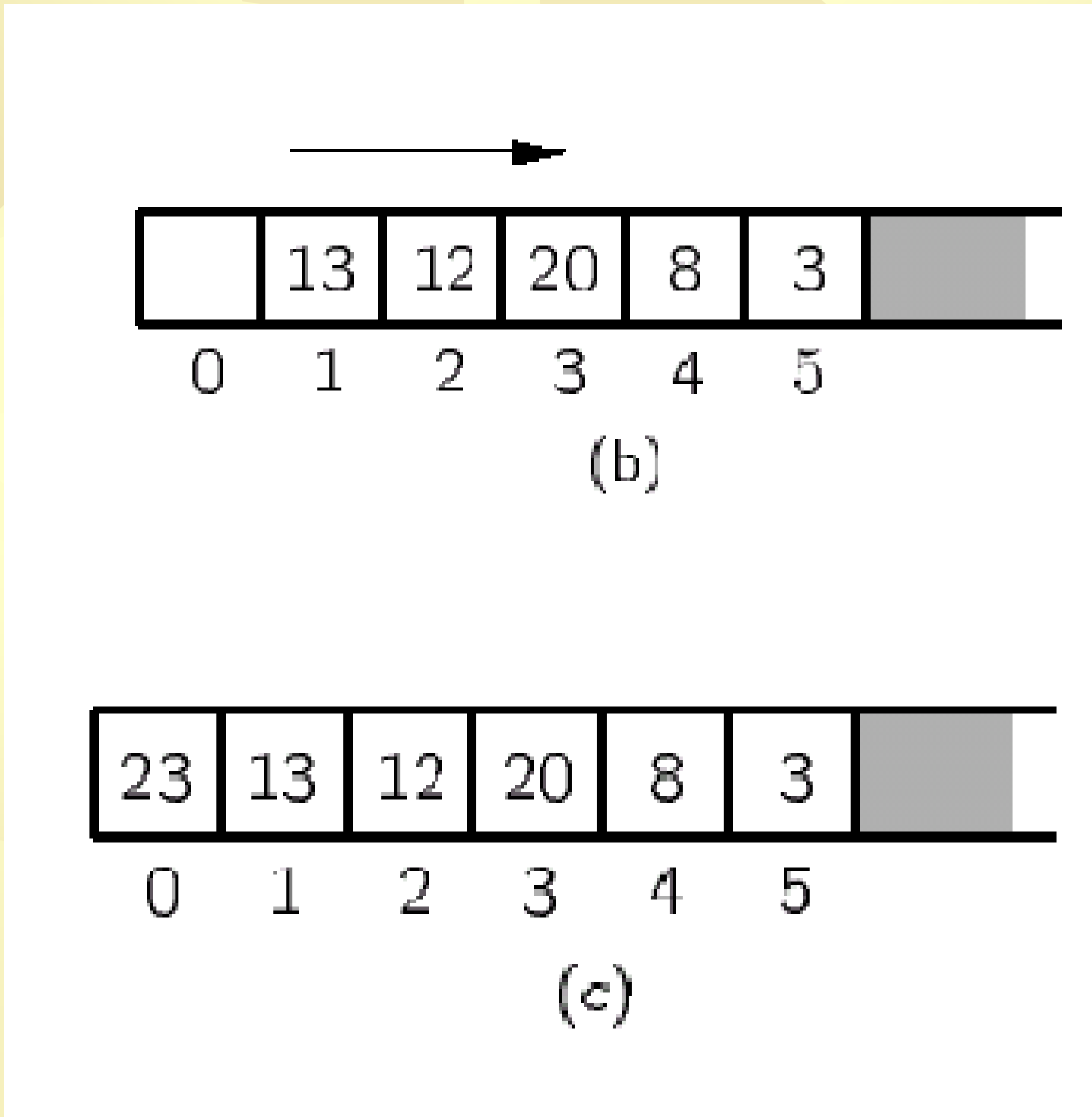
- List: (13; 12; 20 ; 8 ; 3)
  - List: MyList
  - MyList.insert(23);
  - Assume MyPos has 13 as current element

Put 23 before current element,
yielding (23 ; 13; 12; 20 ; 8 ; 3)

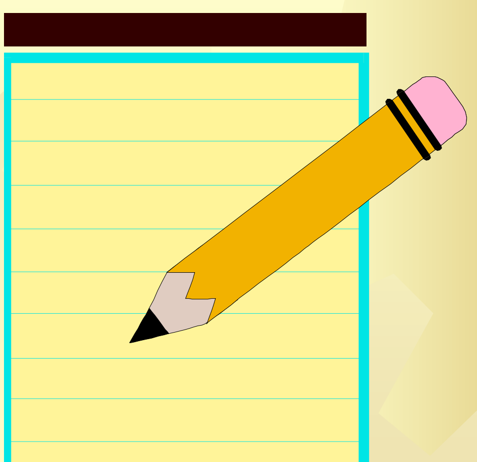
Array-Based List Insert



Array-Based List Insert



Questions?



Class Templates

- A C++ template construction allows us to write code, usually code to implement a class, that uses objects of an arbitrary, generic type.
- In template code we utilize a parameter enclosed in angle brackets < >to denote the generic type.
- template <<parameter list>> class <template name>
  {   <template body>
  };

Class Templates

- Later, when a client uses our code, the client can substitute an actual type for the template parameter. The client can thus obtain several actual pieces of code from our template, using different actual types in place of the template parameter.
- <template name><<actual type>>

Class Templates

- Example: We shall implement a template class List that depends on one generic type parameter. A client can then use our template to declare several lists with different types of entries with declarations of the following form:

```
template <class List_entry> class List {};
List<int> first_list;
List<char> second_list;
```

Class Templates

- Templates provide a new mechanism for creating generic data structures, one that allows many different specializations of a given data structure template in a single application.
- The added generality that we get by using templates comes at the price of slightly more complicated class specifications and implementations.

Implementation of List

- contiguous implementations using arrays
- linked implementations using pointers
  - Simple linked implementation
  - Doubly linked list
  - Circular list

```
template <class List_entry>
class List {
public:
//     methods of the List ADT
    List( );
    int size( ) const;
    bool full( ) const;
    bool empty( ) const;
    void clear( );
    void traverse(void (*visit)(List_entry &));
    Error_code retrieve(int position, List_entry &x) const;
    Error_code replace(int position, const List_entry &x);
    Error_code remove(int position, List_entry &x);
    Error_code insert(int position, const List_entry &x);
```

Number of entries

```
protected:
//    data members for a contiguous list implementation
   int count;
   List_entry entry[max_list];
};
```

Storage

Function Templates

- Many of the methods depend on the template parameter, and so must be implemented as templates too.

- template <<parameter list>> <function definition>;

- Function definition:
  <return type><function name>(<parameter list>)
  {<function body>};

List Size

```
template <class List_entry>
int List<List_entry> :: size( ) const
/* Post: The function returns the number of entries in the List. */
{
   return count;
}
```

Insertion

template <**class** List_entry>
Error_code List<List_entry> :: insert(**int** position, **const** List_entry &x)
/* **Post:** *If the* List *is not full and* $0 \leq$ position $\leq n$, *where* n *is the number of entries in the* List, *the function succeeds: Any entry formerly at* position *and all later entries have their position numbers increased by 1 and* **x** *is inserted at* position *of the* List.
Else: The function fails with a diagnostic error code. */

```
{
  if (full( ))
     return overflow;

  if (position < 0 || position > count)
     return range_error;

  for (int i = count - 1; i >= position; i--)
     entry[i + 1] = entry[i];

  entry[position] = x;
  count++;
  return success;
}
```

Traversal

```
template <class List_entry>
void List<List_entry> :: traverse(void (*visit)(List_entry &))
/* Post: The action specified by function (*visit) has been performed on every entry of
         the List, beginning at position 0 and doing each in turn. */
{
  for (int i = 0; i < count; i++)
    (*visit)(entry[i]);
}
```

Performance of Methods

- In processing a contiguous list with n entries:
  - insert and remove operate in time approximately proportional to n.
  - List, clear, empty, full, size, replace, and retrieve operate in constant time.

Simple linked implementation

Node declaration:

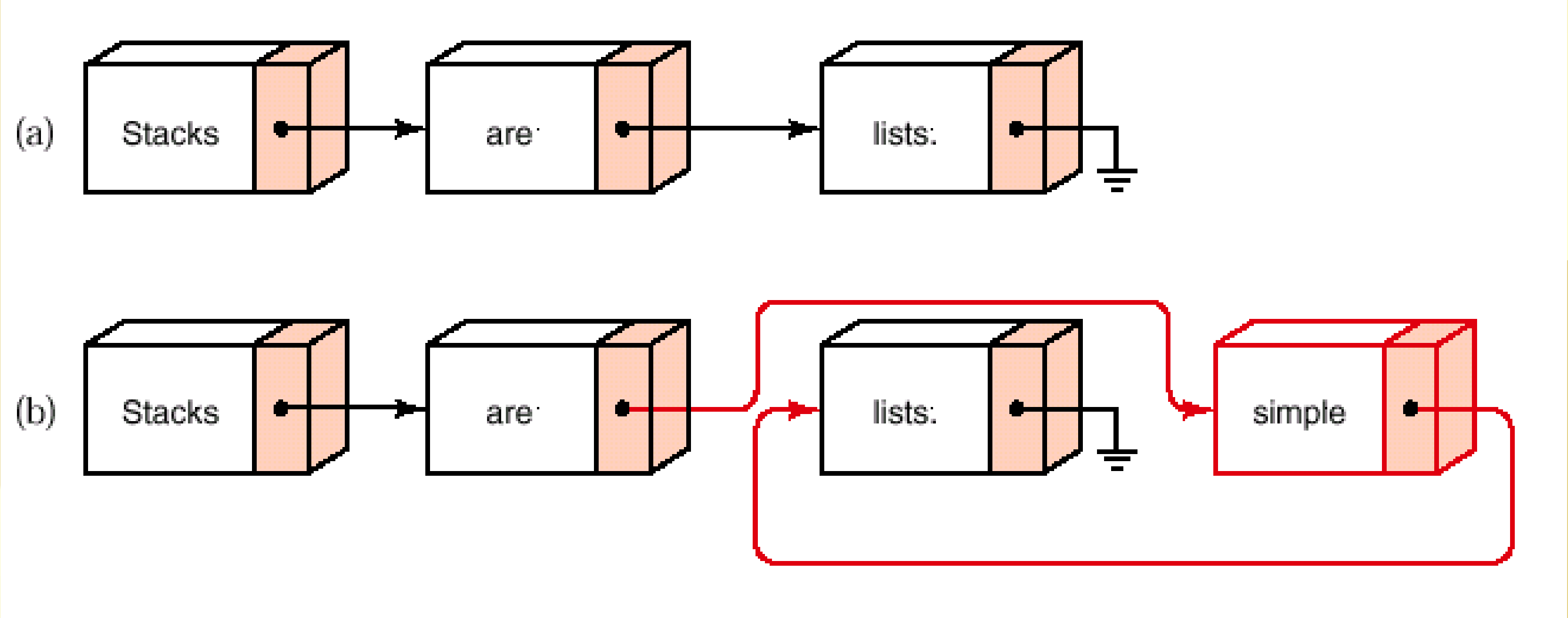
```
template <class Node_entry>
struct Node {
//     data members
   Node_entry entry;
   Node<Node_entry> *next;
//     constructors
   Node( );
   Node(Node_entry, Node<Node_entry> *link = NULL);
};
```

List declaration:

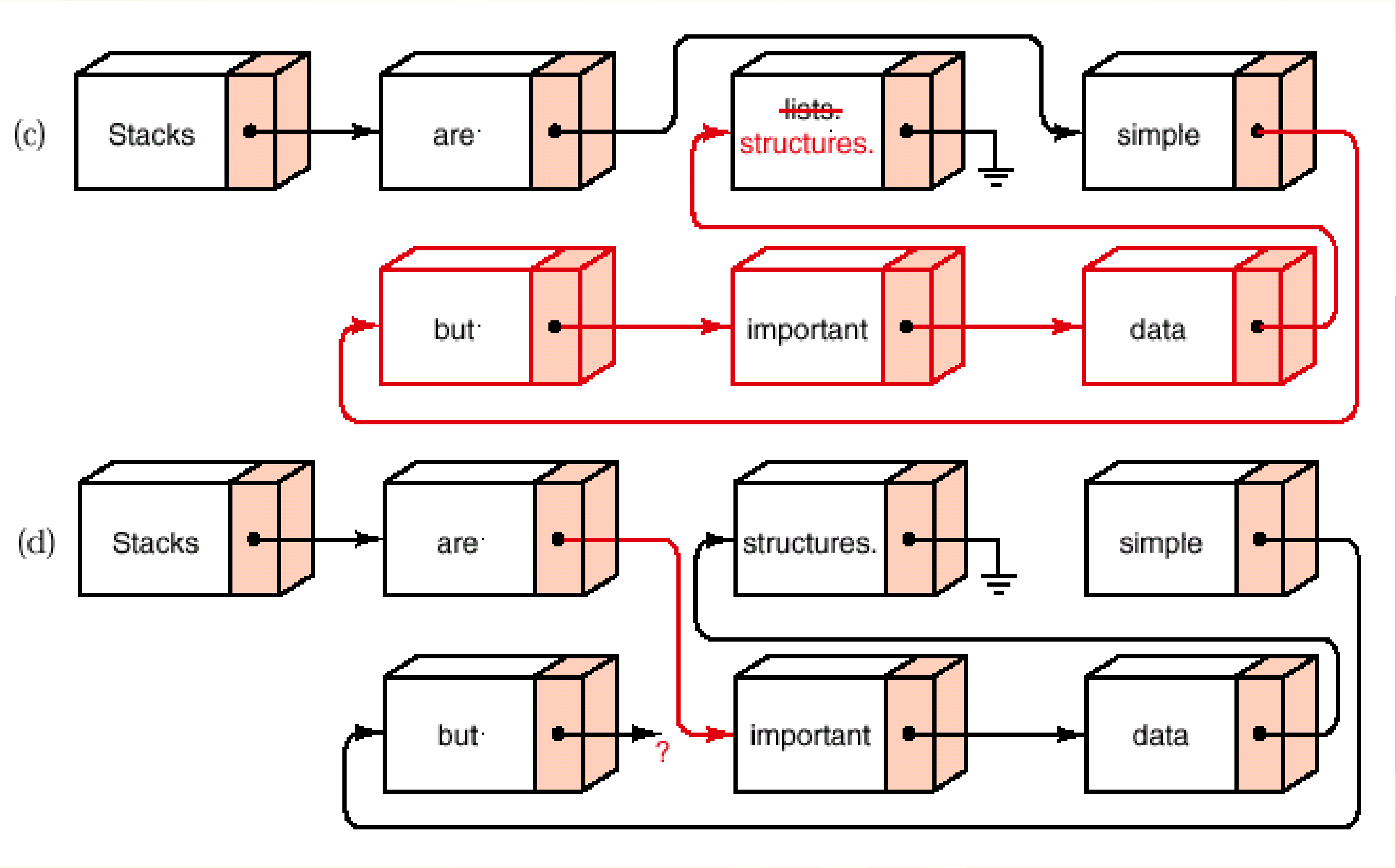
```
template <class List_entry>
class List {
public:
//    Specifications for the methods of the list ADT go here.

//    The following methods replace compiler-generated defaults.
    ~List( );
   List(const List<List_entry> &copy);
   void operator = (const List<List_entry> &copy);
protected:
//    Data members for the linked list implementation now follow.
   int count;
   Node<List_entry> *head;
//    The following auxiliary function is used to locate list positions
   Node<List_entry> *set_position(int position) const;
};
```

Actions on a Linked List



Actions on a Linked List



Finding a List Position

- Function set_position takes an integer parameter position and returns a pointer to the corresponding node of the list.
- Declare the visibility of set_position as protected, since set_position returns a pointer to, and therefore gives access to, a Node in the List.

Finding a List Position

- To construct set_position, we start at the beginning of the List and traverse it until we reach the desired node.
- If all nodes are equally likely, then, on average, the set position function must move halfway through the List to find a given position.
- On average, its time requirement is approximately proportional to n, the size of the List.

Implementation of set_position

```
template <class List_entry>
Node<List_entry> *List<List_entry> :: set_position(int position) const
/* Pre:   position is a valid position in the List; 0 ≤ position < count.
   Post:  Returns a pointer to the Node in position.  */
{
   Node<List_entry> *q = head;
   for (int i = 0;  i < position;  i++) q = q->next;
   return q;
}
```

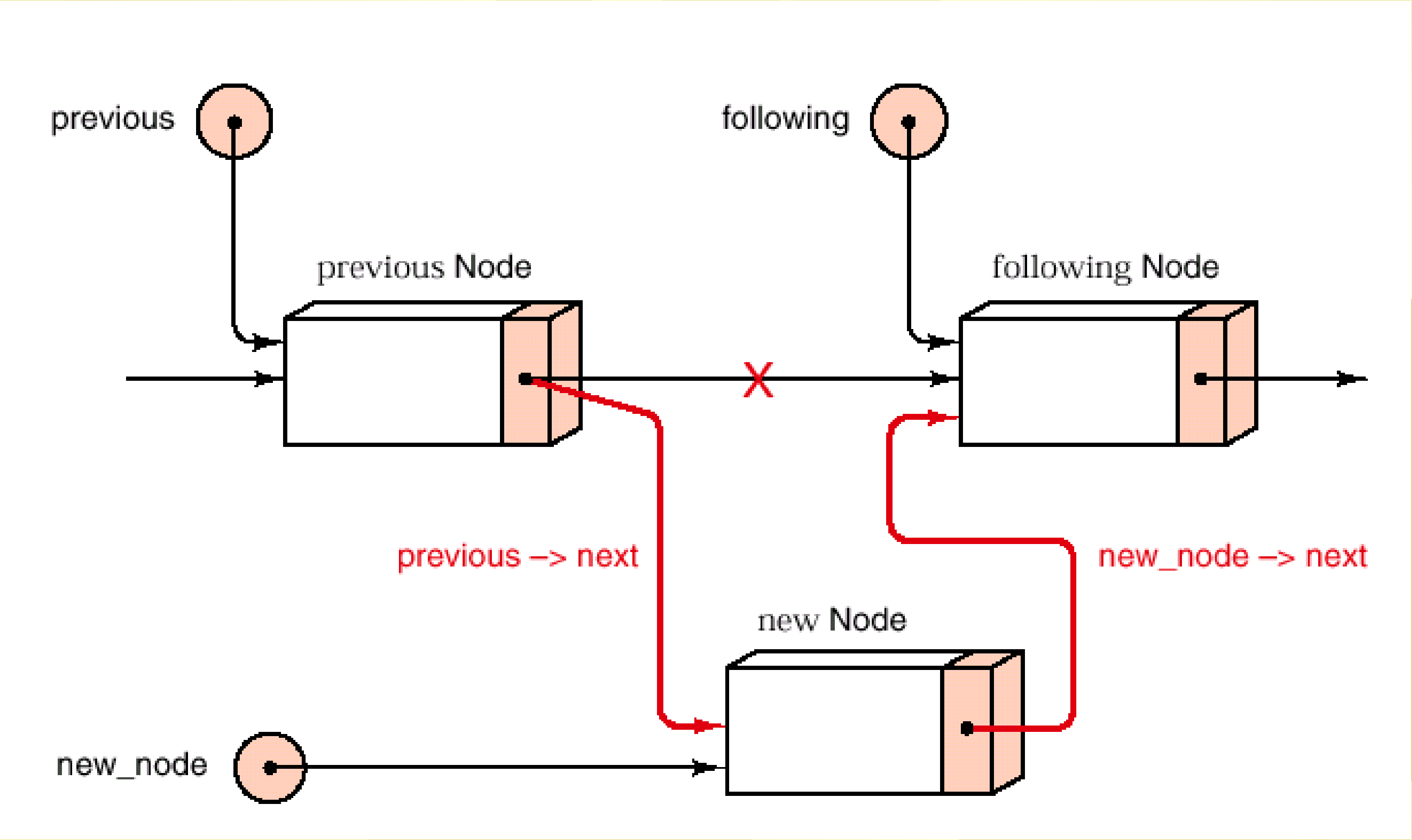
Insertion

template <**class** List_entry>
Error_code List<List_entry> :: insert(**int** position, **const** List_entry &x)
/* **Post:** *If the* List *is not full and* $0 \leq$ position $\leq n$, *where* n *is the number of entries in the* List, *the function succeeds: Any entry formerly at* position *and all later entries have their position numbers increased by 1 and* x *is inserted at* position *of the* List.
Else: The function fails with a diagnostic error code. */

```
{
   if (position < 0 || position > count)
      return range_error;
   Node<List_entry> *new_node, *previous, *following;
   if (position > 0) {
      previous = set_position(position − 1);
      following = previous->next;
   }
   else following = head;
```

```
new_node = new Node<List_entry>(x, following);
if (new_node == NULL)
    return overflow;
if (position == 0)
    head = new_node;
else
    previous->next = new_node;
count++;
return success;
}
```

Insertion into a Linked List



Performance of Methods

- In processing a linked List with n entries
  - clear, insert, remove, retrieve, and replace require time approximately proportional to n.
  - List, empty, full, and size operate in constant time.

The cost

- The cost of a list is the extra space required in each Node for a link
  - E: Space for data value
  - P: Space for pointer
  - D: Number of elements in array
  - n: Count of List

Keeping the Current Position

- Suppose an application processes list entries in order or refers to the same entry several times before processing another entry.
- Remember the last-used position in the list and, if the next operation refers to the same or a later position, start tracing through the list from this last-used position.

```
template <class List_entry>
class List {
public:
//    Add specifications for the methods of the list ADT.
//    Add methods to replace the compiler-generated defaults.

protected:
//    Data members for the linked-list implementation with
//    current position follow:
   int count;
   mutable int current_position;
   Node<List_entry> *head;
   mutable Node<List_entry> *current;

//    Auxiliary function to locate list positions follows:
   void set_position(int position) const;
};
```

- Add current_position
- The current_position is now a member of the class List, so there is no longer a need for set position to return a pointer; instead, the function simply resets the pointer current directly within the List.

Implementation of set_position

```
template <class List_entry>
void List<List_entry> :: set_position(int position) const
/* Pre:   position is a valid position in the List: 0 ≤ position < count.
   Post:  The current Node pointer references the Node at position.  */
{
   if (position < current_position) {   //   must start over at head of list
      current_position = 0;
      current = head;
   }
   for ( ; current_position != position; current_position++)
      current = current->next;
}
```

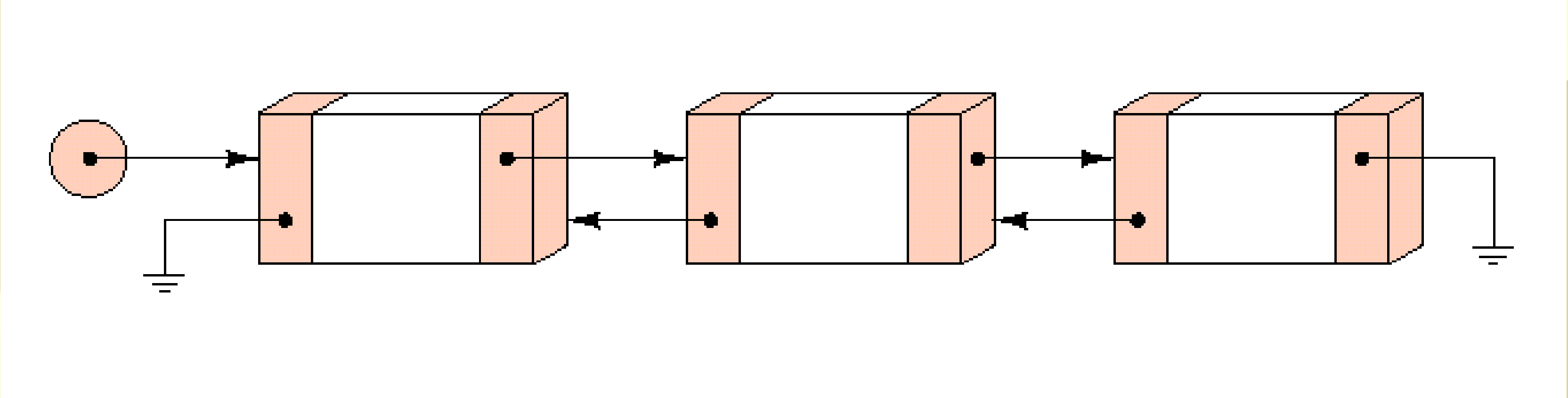
Performance of Methods

- For repeated references to the same position, neither the body of the if statement nor the body of the for statement will be executed, and hence the function will take almost no time.

Performance of Methods

- If we move forward only one position, the body of the for statement will be executed only once, so again the function will be very fast.
- If it is necessary to move backwards through the List, then the function operates in almost the same way as the version of set position used in the previous implementation.

Doubly Linked Lists



Node definition

```
template <class Node_entry>
struct Node {
//    data members
   Node_entry entry;
   Node<Node_entry> *next;
   Node<Node_entry> *back;
//    constructors
   Node( );
   Node(Node_entry, Node<Node_entry> *link_back = NULL,
                    Node<Node_entry> *link_next = NULL);
};
```

List definition

```
template <class List_entry>
class List {
public:
//    Add specifications for methods of the list ADT.
//    Add methods to replace compiler generated defaults.

protected:
//    Data members for the doubly-linked list implementation follow:
    int count;
    mutable int current_position;
    mutable Node<List_entry> *current;


//    The auxiliary function to locate list positions follows:
    void set_position(int position) const;
};
```

Doubly Linked Lists

- We can move either direction through the List while keeping only one pointer, current, into the List.
- We do not need pointers to the head or the tail of the List, since they can be found by tracing back or forth from any given node.

Doubly Linked Lists

- To find any position in the doubly linked list, we first decide whether to move forward or backward from the current position,and then we do a partial traversal of the list until we reach the desired position.

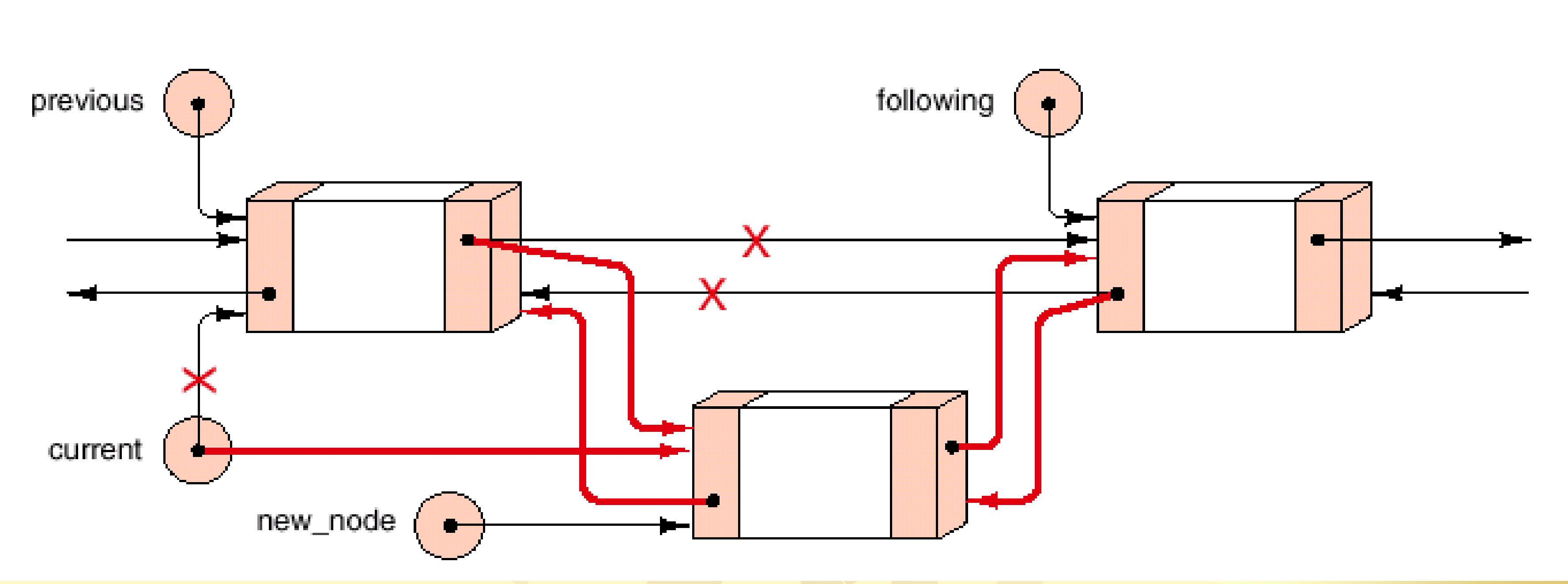
Implementation of set_position

```
template <class List_entry>
void List<List_entry> :: set_position(int position) const
/* Pre:   position is a valid position in the List: 0 ≤ position < count.
   Post:  The current Node pointer references the Node at position.  */
{
  if (current_position <= position)
    for ( ; current_position != position; current_position++)
      current = current->next;
  else
    for ( ; current_position != position; current_position--)
      current = current->back;
}
```

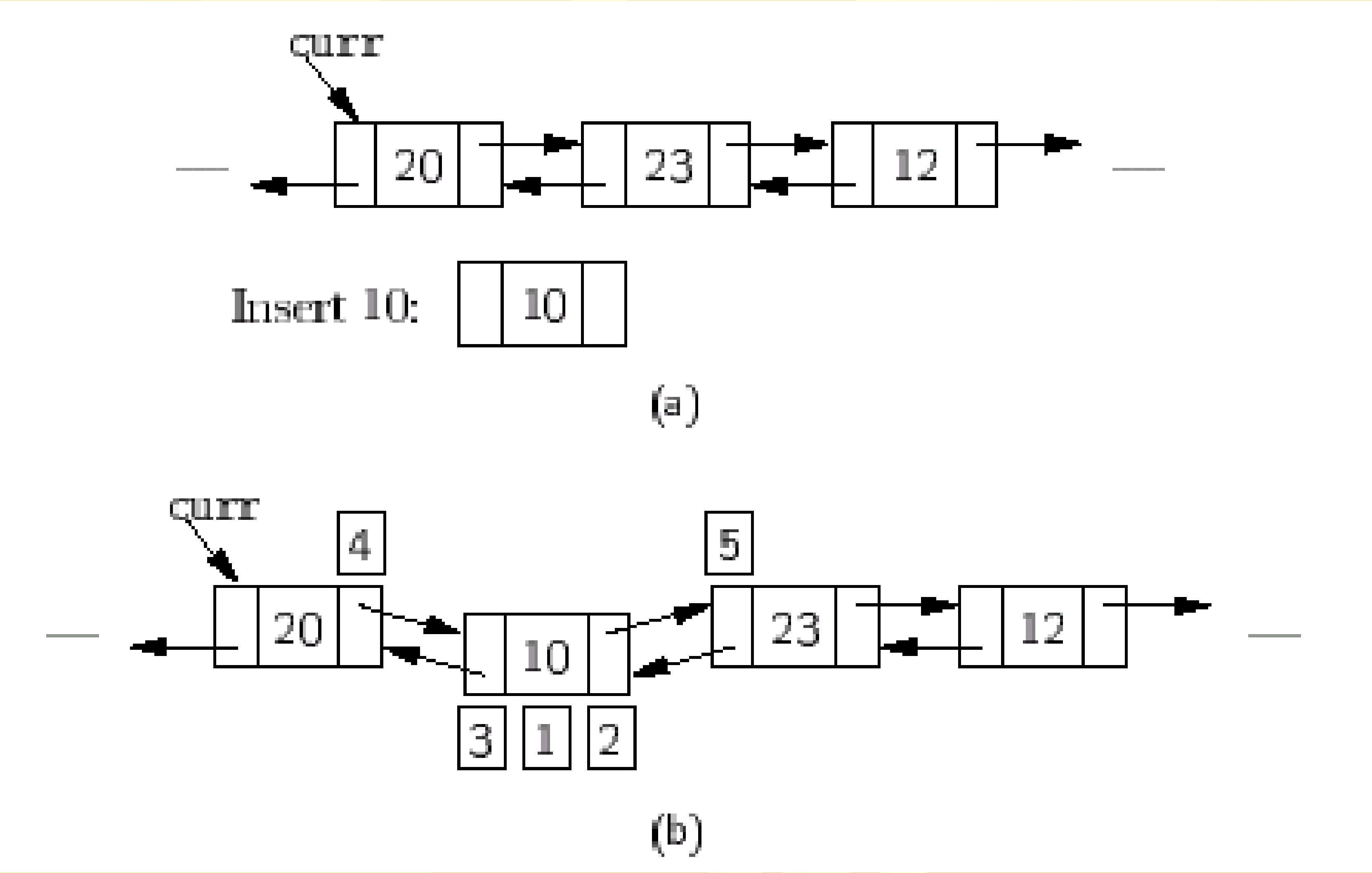
The cost

- The cost of a doubly linked list is the extra space required in each Node for a second link, usually trivial in comparison to the space for the information member entry.

Insertion into a Doubly Linked List



Steps of Insert



Insertion into a Doubly Linked List

```
template <class List_entry>
Error_code List<List_entry> :: insert(int position, const List_entry &x)
/* Post: If the List is not full and 0 ≤ position ≤ n, where n is the number of
         entries in the List, the function succeeds: Any entry formerly at position and
         all later entries have their position numbers increased by 1 and x is inserted at
         position of the List.
         Else: the function fails with a diagnostic error code. */

{
  Node<List_entry> *new_node, *following, *preceding;

  if (position < 0 || position > count) return range_error;
```

Insertion into a Doubly Linked List

```
if (position == 0) {
   if (count == 0) following = NULL;
   else {
      set_position(0);
      following = current;
   }
   preceding = NULL;
}

else {
   set_position(position − 1);
   preceding = current;
   following = preceding->next;
}
new_node = new Node<List_entry>(x, preceding, following);
```

Insertion into a Doubly Linked List

```
    if (new_node == NULL) return overflow;
    if (preceding != NULL) preceding->next = new_node;
    if (following != NULL) following->back = new_node;
    current = new_node;
    current_position = position;
    count++;
    return success;
}
```

Comparison of Implementations

- Contiguous storage is generally preferable
  - when the entries are individually very small;
  - when the size of the list is known when the program is written;
  - when few insertions or deletions need to be made except at the end of the list; and
  - when random access is important.

Comparison of Implementations

- Linked storage proves superior
  - when the entries are large;
  - when the size of the list is not known in advance; and
  - when flexibility is needed in inserting, deleting, and rearranging the entries.

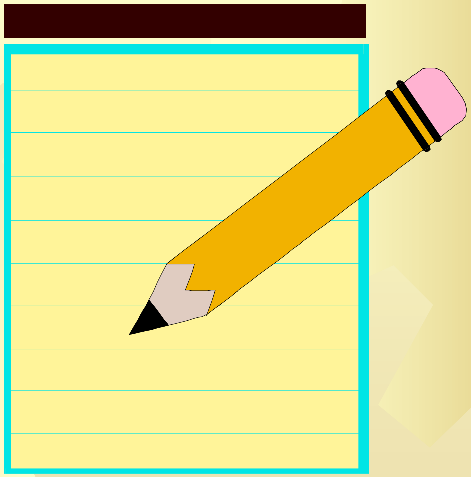
How to choose

- Which of the operations will actually be performed on the list and which of these are the most important?
- Is there locality of reference? That is, if one entry is accessed, is it likely that it will next be accessed again?

How to choose

- Are the entries processed in sequential order? If so, then it may be worthwhile to maintain the last-used position as part of the list structure.
- Is it necessary to move both directions through the list? If so, then doubly linked lists may prove advantageous.

Questions?

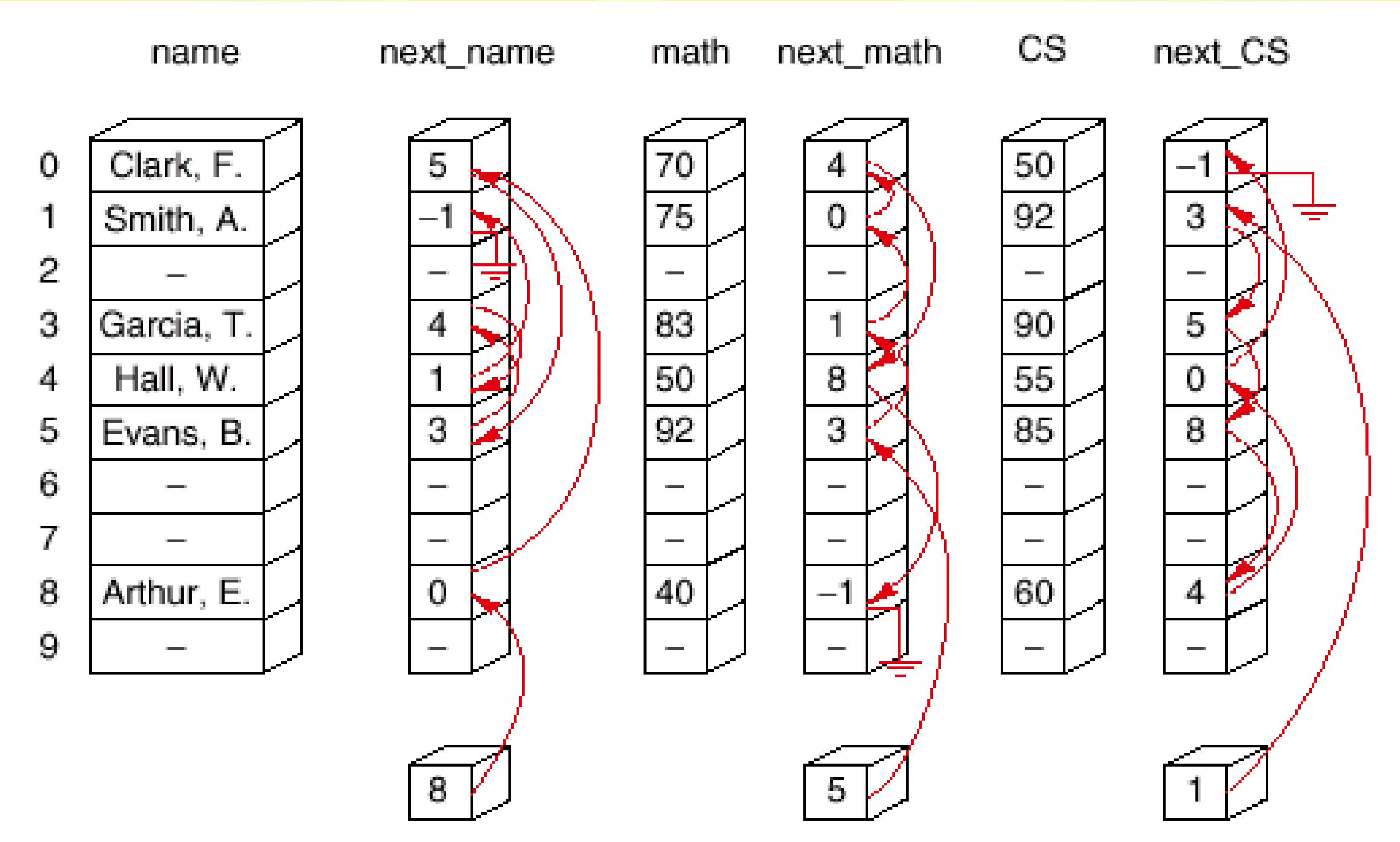


Linked Lists in Arrays

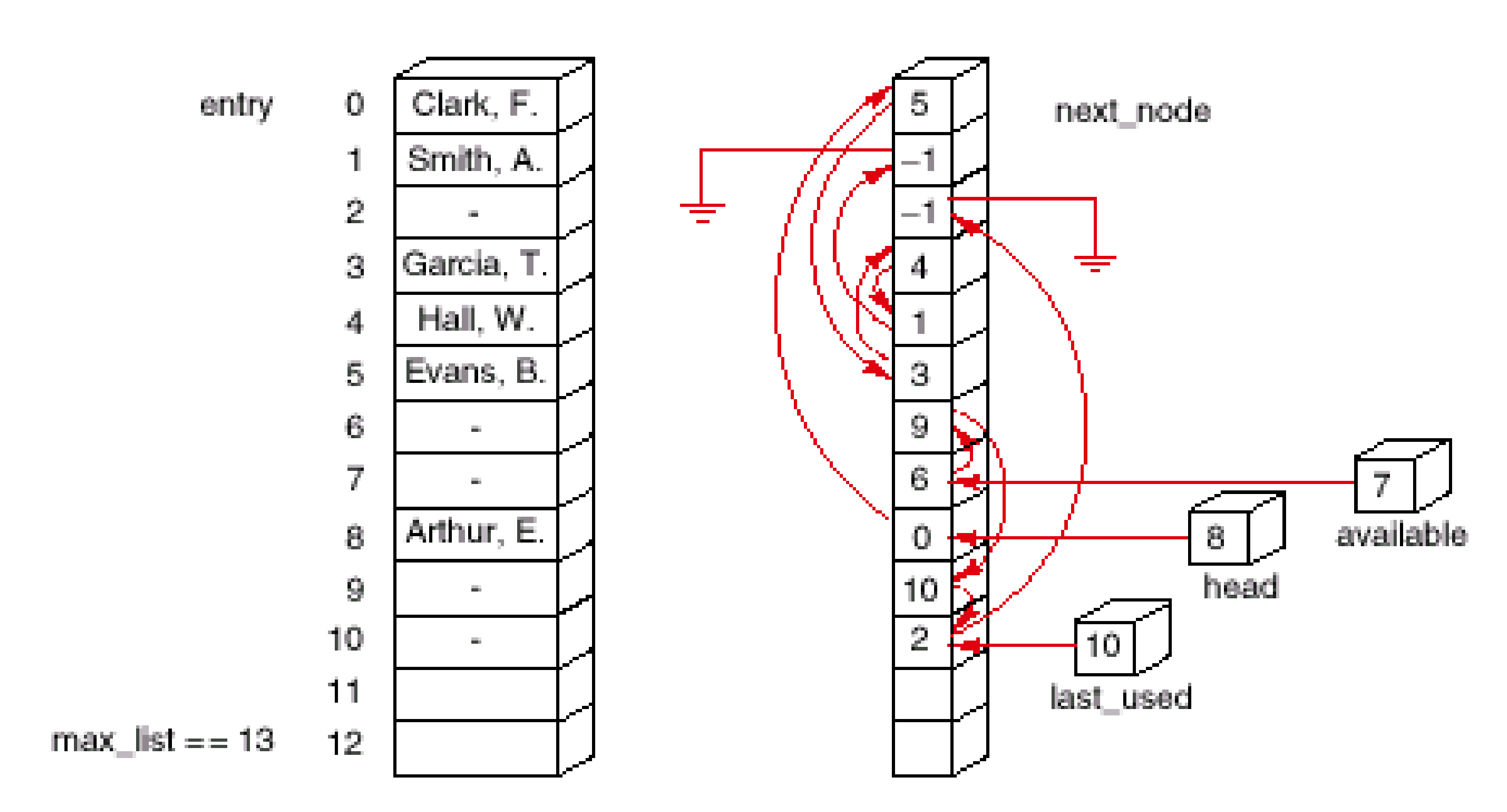
- Applications where linked lists in arrays may prove preferable are those where
  - the number of entries in a list is known in advance,
  - the links are frequently rearranged, but relatively few additions or deletions are made, or
  - the same data are sometimes best treated as a linked list and other times as a contiguous list.

	name	math	CS
0	Clark, F.	70	50
1	Smith, A.	75	92
2			
3	Garcia, T.	83	90
4	Hall, W.	50	55
5	Evans, B.	92	85
6			
7			
8	Arthur, E.	40	60
9			

Linked Lists in Arrays



Linked Lists in Arrays



Class Declaration

```
template <class List_entry>
class Node {
public:
    List_entry entry;
    index next;
};
```

Class Declaration

```
template <class List_entry>
class List {
public:
//    Methods of the list ADT
    List( );
    int size( ) const;
    bool full( ) const;
    bool empty( ) const;
    void clear( );
    void traverse(void (*visit)(List_entry &));
    Error_code retrieve(int position, List_entry &x) const;
    Error_code replace(int position, const List_entry &x);
    Error_code remove(int position, List_entry &x);
    Error_code insert(int position, const List_entry &x);
```

Class Declaration

```
protected:
//   Data members
   Node<List_entry> workspace[max_list];
   index available, last_used, head;
   int count;
//   Auxiliary member functions
   index new_node( );
   void delete_node(index n);
   int current_position(index n) const;
   index set_position(int position) const;
};
```

New

```
template <class List_entry>
index List<List_entry> :: new_node( )
/* Post: The index of the first available Node in workspace is returned; the data
         members available, last_used, and workspace are updated as necessary.
         If the workspace is already full, −1 is returned.  */
{
   index new_index;
   if (available != −1) {
      new_index = available;
      available = workspace[available].next;
   } else if (last_used < max_list − 1) {
      new_index = ++last_used;
   } else return −1;
   workspace[new_index].next = −1;
   return new_index;
}
```

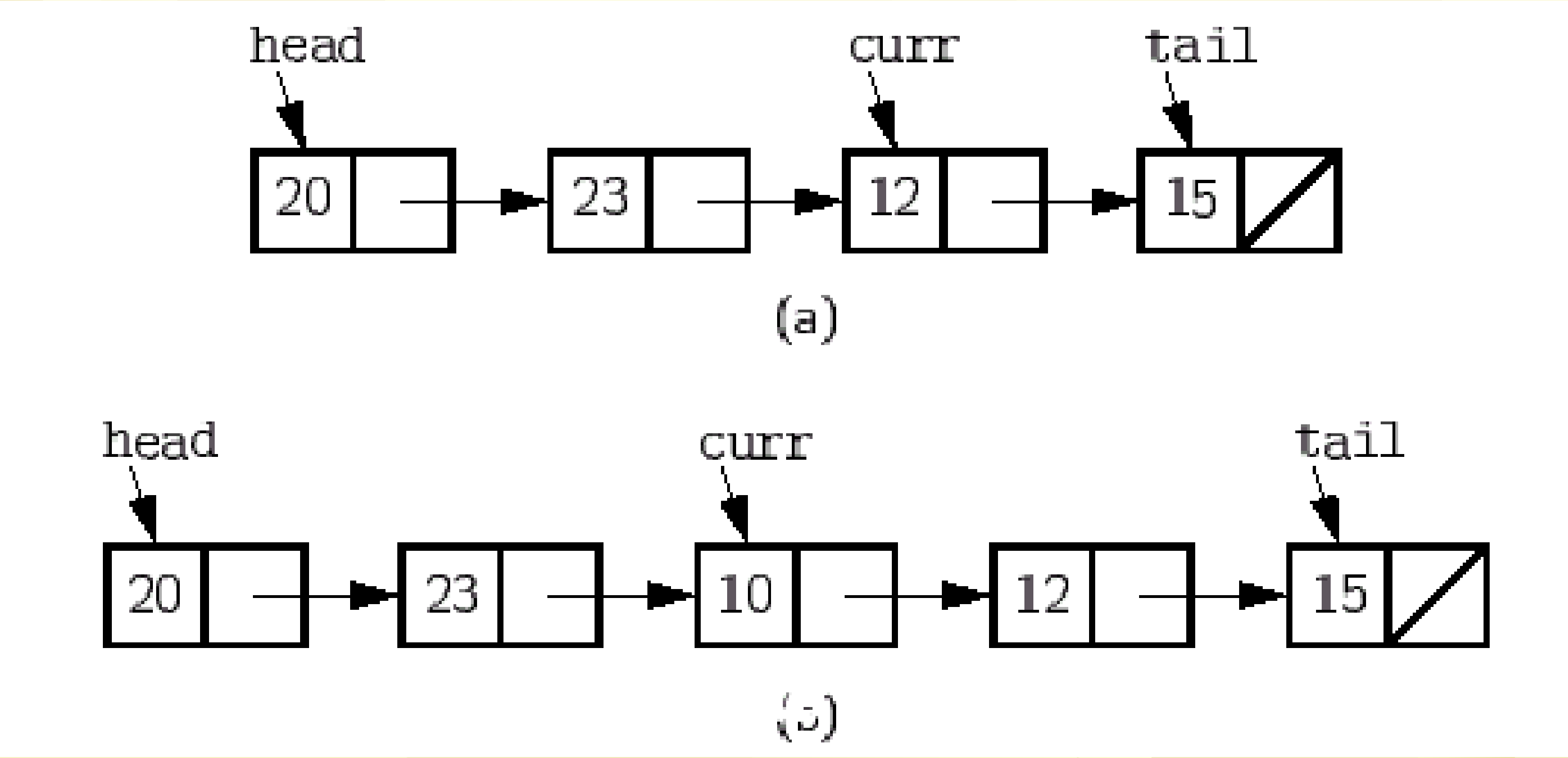
Delete

```
template <class List_entry>
void List<List_entry> :: delete_node(index old_index)
/* Pre:   The List has a Node stored at index old_index.
   Post:  The List index old_index is pushed onto the linked stack of available space;
          available, last_used, and workspace are updated as necessary.  */
{
   index previous;
   if (old_index ==  head) head = workspace[old_index].next;
   else {
      previous = set_position(current_position(old_index) − 1);
      workspace[previous].next = workspace[old_index].next;
   }
   workspace[old_index].next = available;
   available = old_index;
}
```

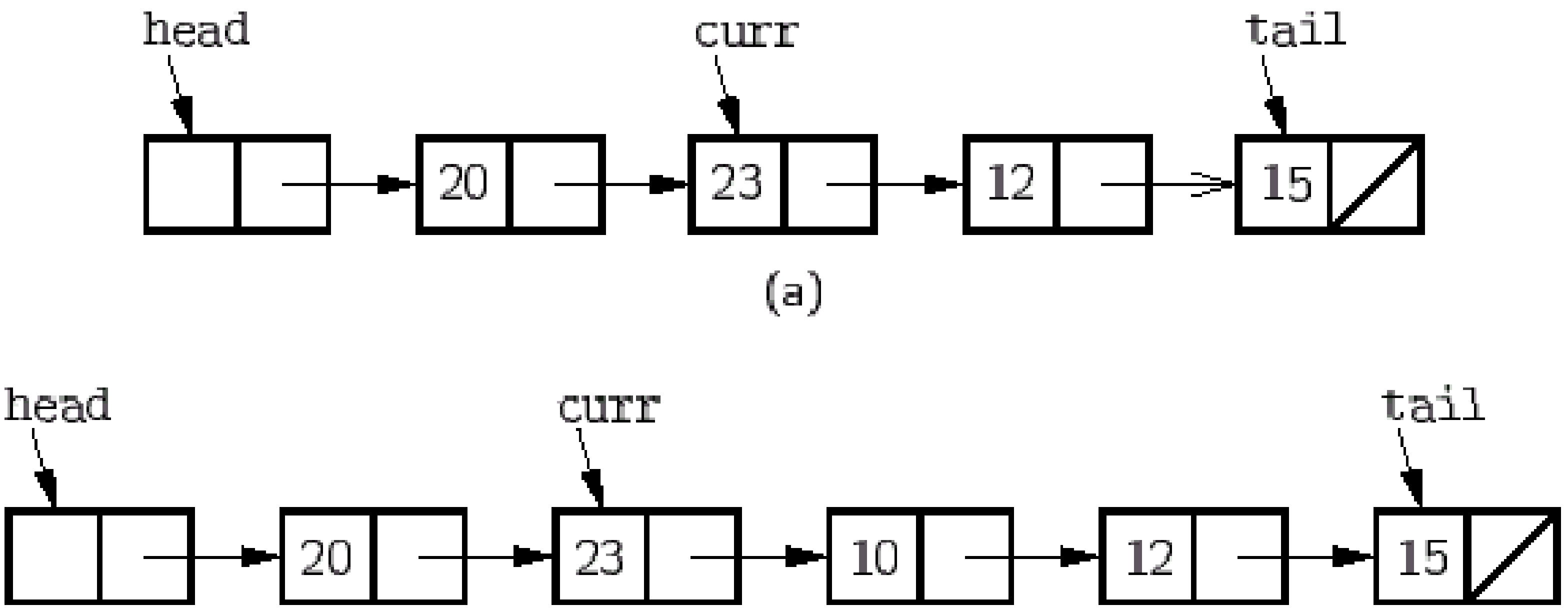
Other Operations

- index List<List entry> ::set_position(int position) const;
- int List<List entry> ::current_position(index n) const;
- void List<List entry> ::traverse(void (*visit)(List entry &))
- Error_code List<List entry> ::insert(int position, const List entry &x)

Example of Insert

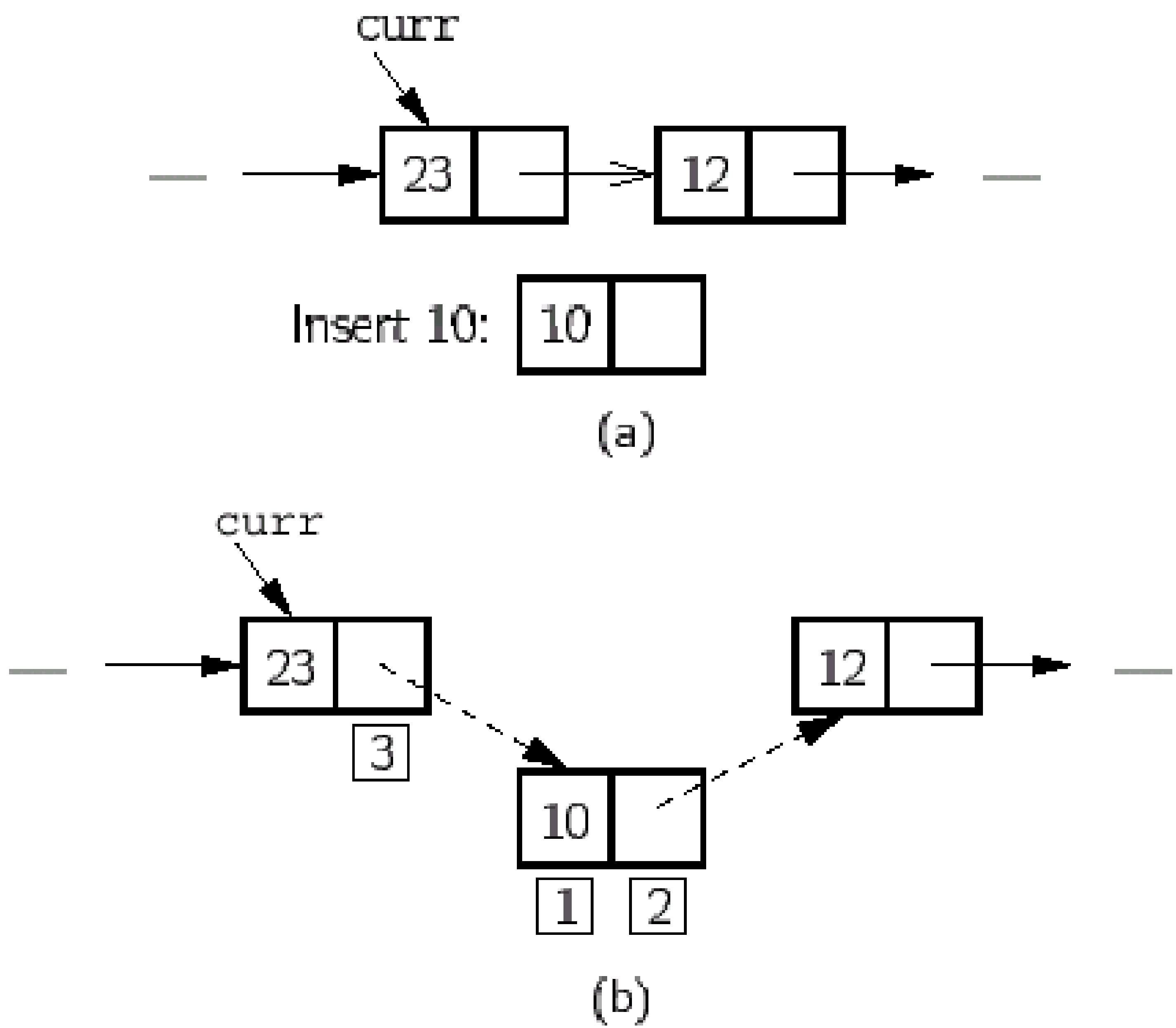


Example of Insert



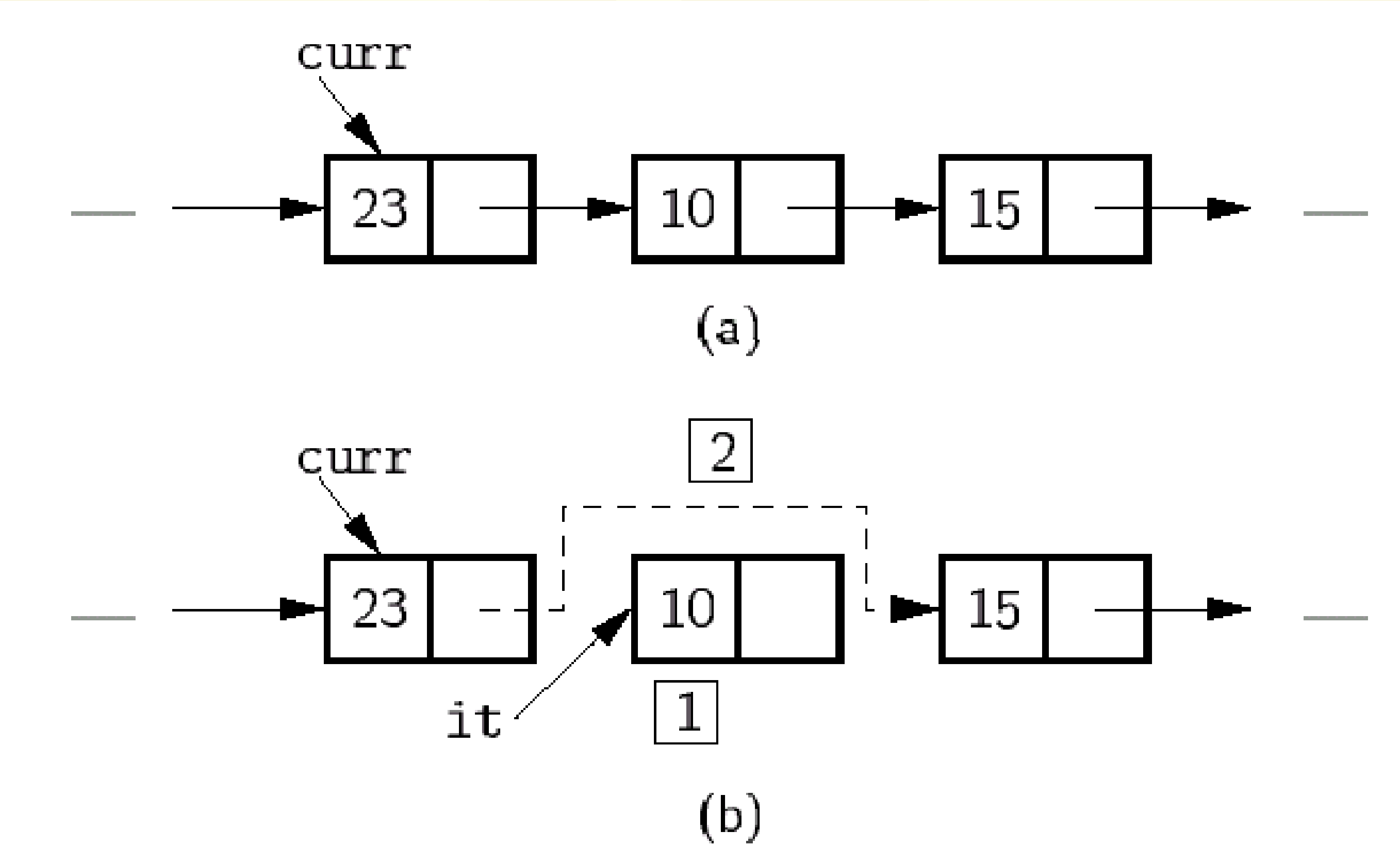
(a)

(b)



curr
23
12
Insert 10:
10
(a)
curr
23
12
3
10
1
2
(b)

Example of Remove



FreeList

- System new and delete are slow