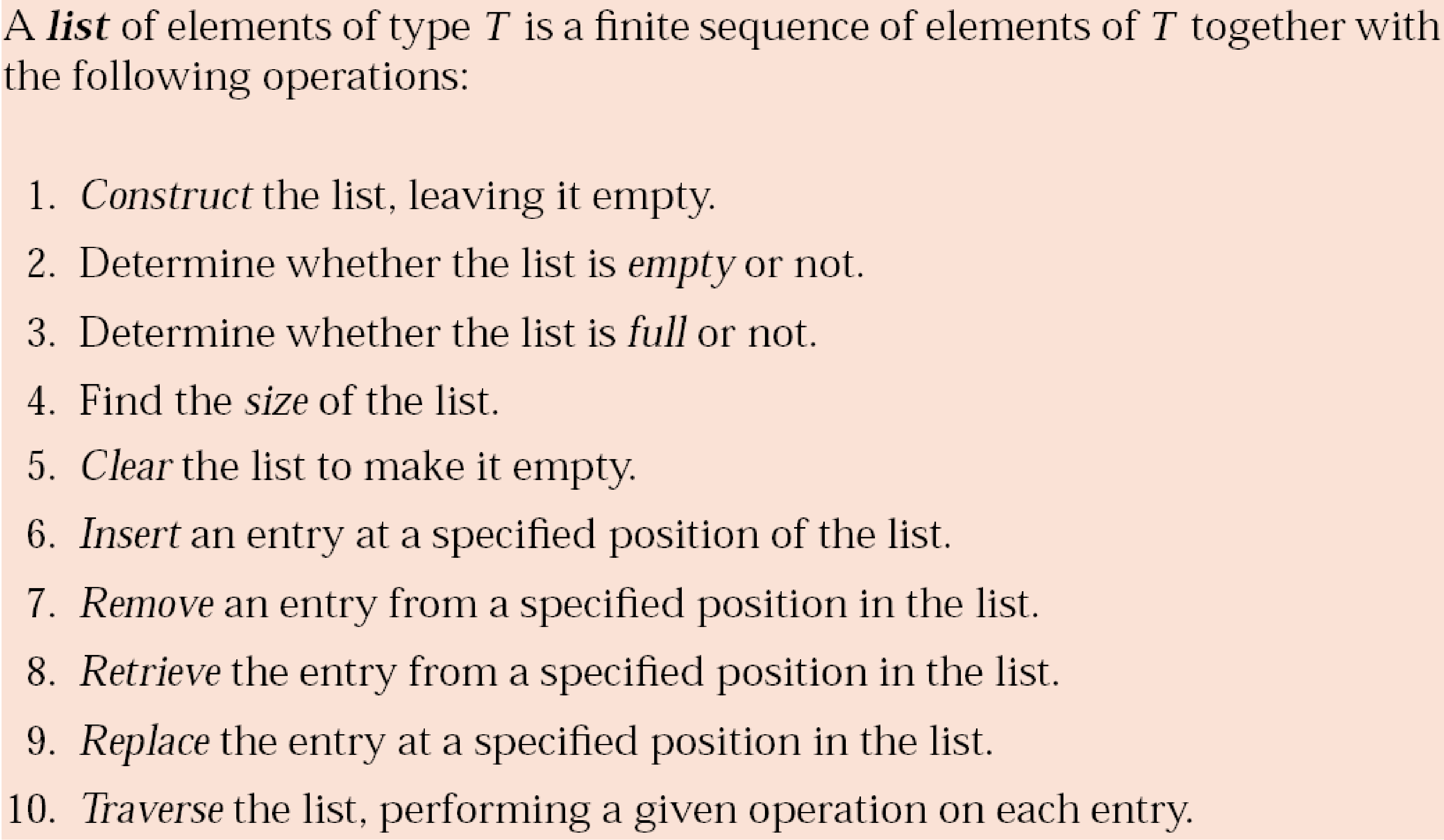
Chapter 6 Lists and Strings

信息科学与技术学院　　黄方军

data_structures@163.com

东校区实验中心B502

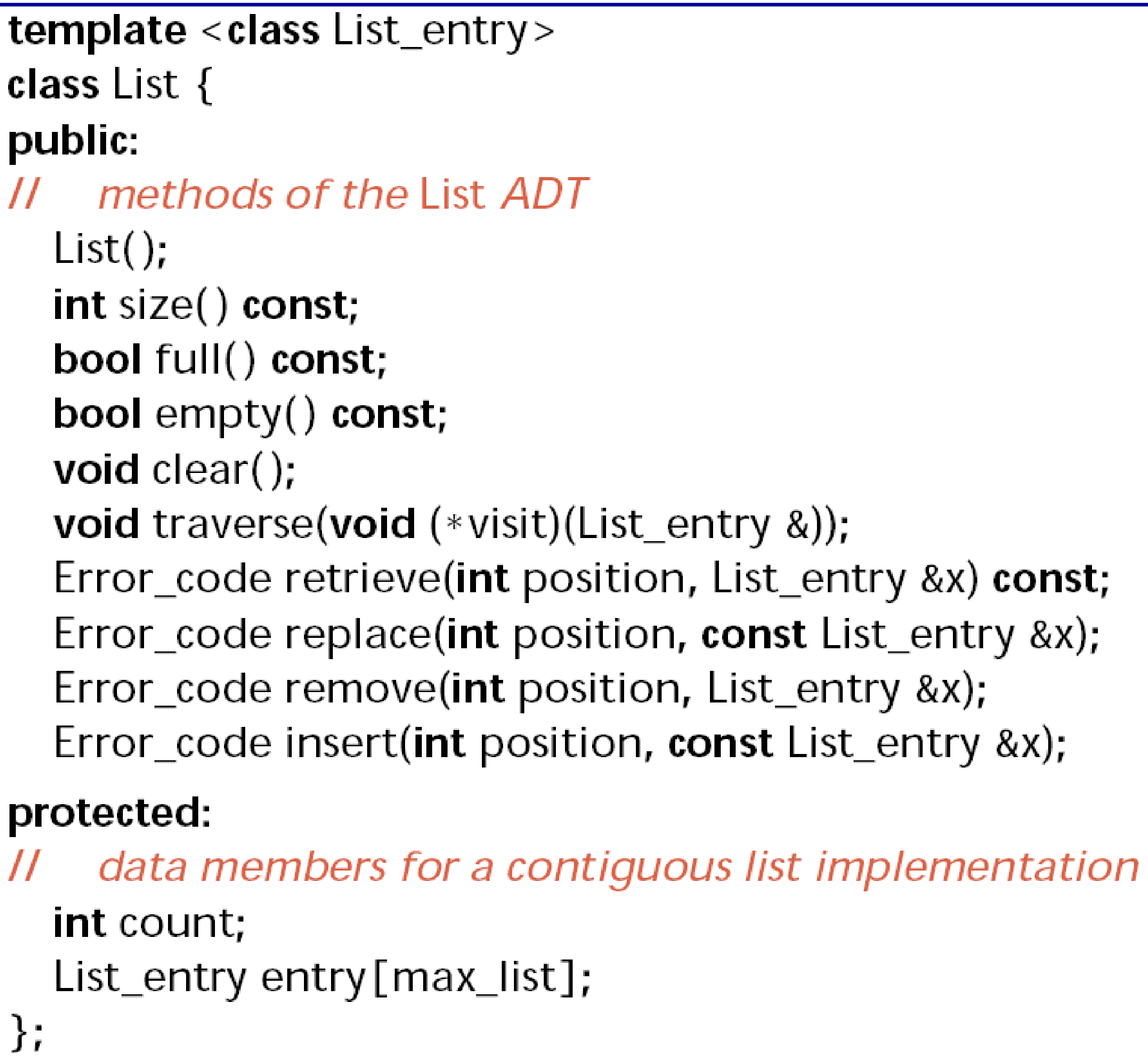
6.1 List Definition



A ***list*** of elements of type T is a finite sequence of elements of T together with the following operations:

1. *Construct* the list, leaving it empty.
2. Determine whether the list is *empty* or not.
3. Determine whether the list is *full* or not.
4. Find the *size* of the list.
5. *Clear* the list to make it empty.
6. *Insert* an entry at a specified position of the list.
7. *Remove* an entry from a specified position in the list.
8. *Retrieve* the entry from a specified position in the list.
9. *Replace* the entry at a specified position in the list.
10. *Traverse* the list, performing a given operation on each entry.

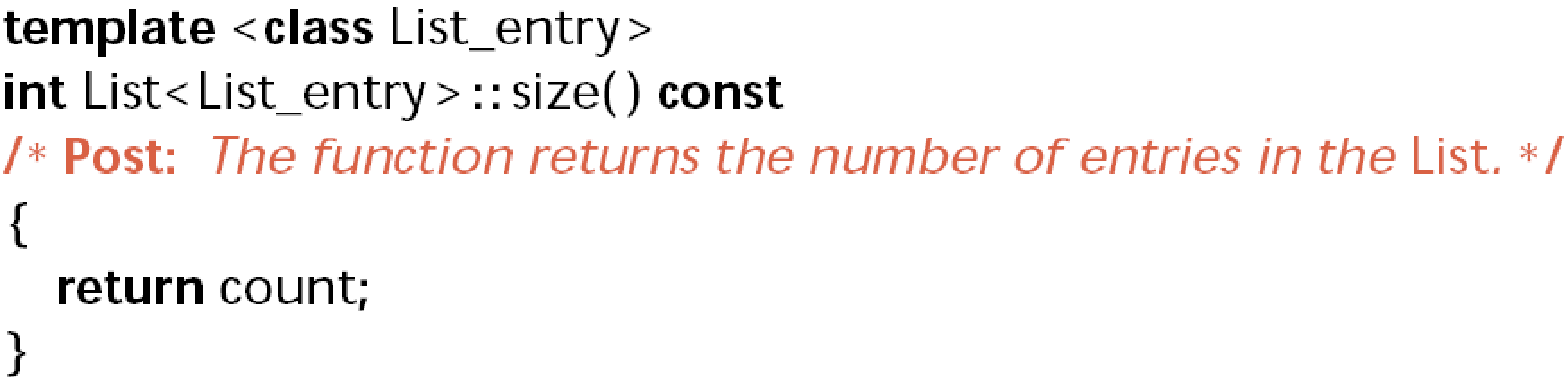
6.2.2 Contiguous Implementation



```
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);

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

6.2.2 Contiguous Implementation

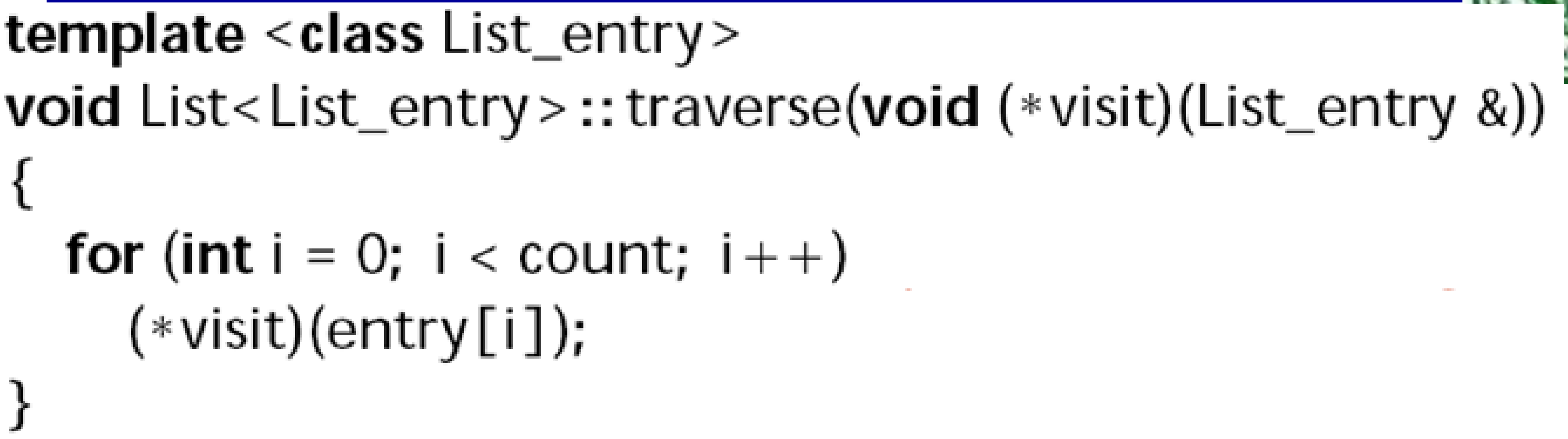


```
template <class List_entry>
int List<List_entry>::size() const
/* Post:  The function returns the number of entries in the List. */
{
   return count;
}
```

6.2.2 Contiguous Implementation

```
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. */
{
   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;
}
```

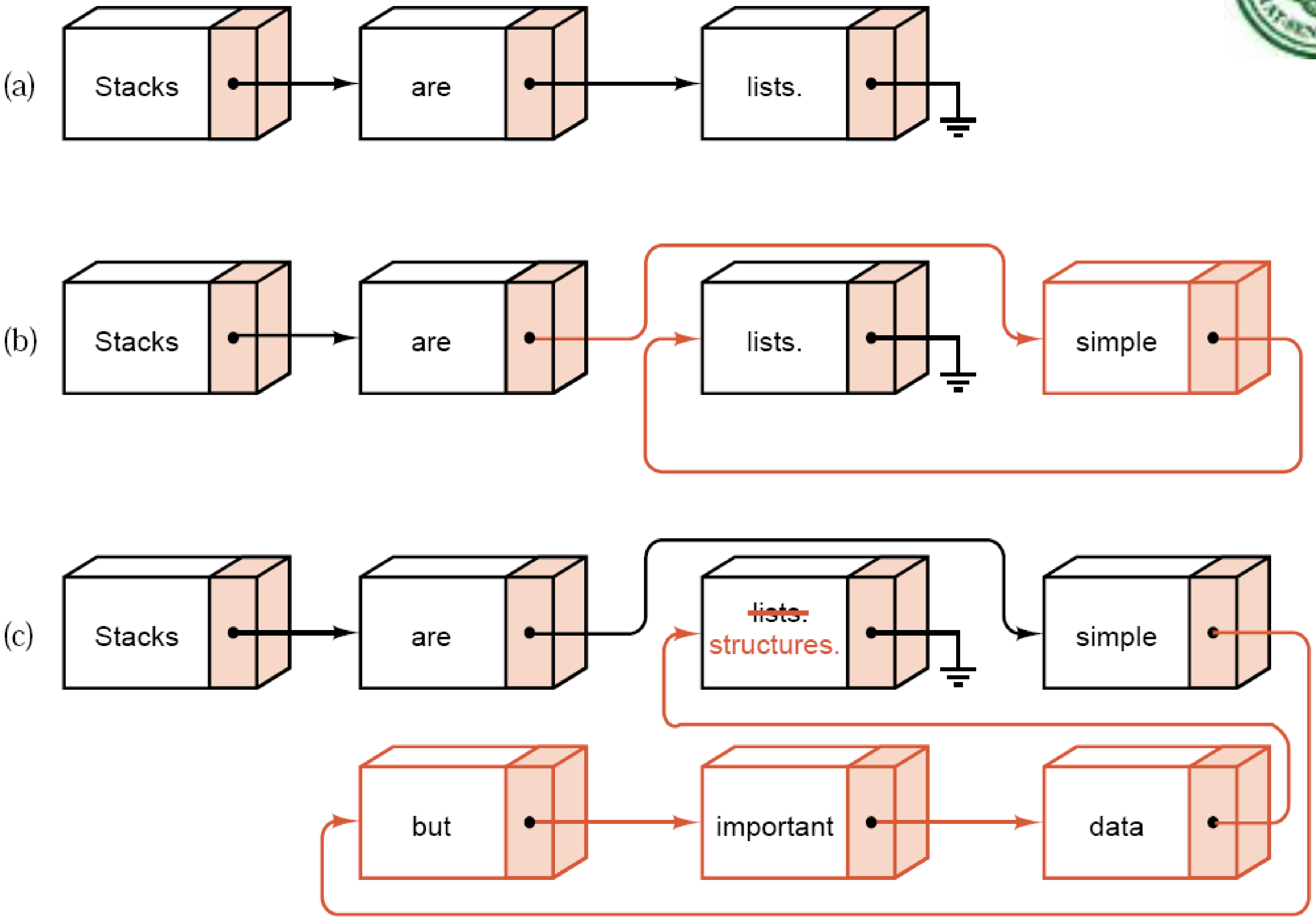
6.2.2 Contiguous Implementation



```
template <class List_entry>
void List<List_entry> :: traverse(void (*visit)(List_entry &))
{
   for (int i = 0; i < count; i++)
      (*visit)(entry[i]);
}
```

```
void write_entry(char &c)
{
   cout << c;
}
int main()
{  char x;
   List<char> c_list;  //  a list of characters, initialized empty
   c_list.insert(c_list.size(), x);
   c_list.traverse(write_entry);       }
```

6.2.3 Simply Linked Implementation



6.2.3 Simply Linked Implementation

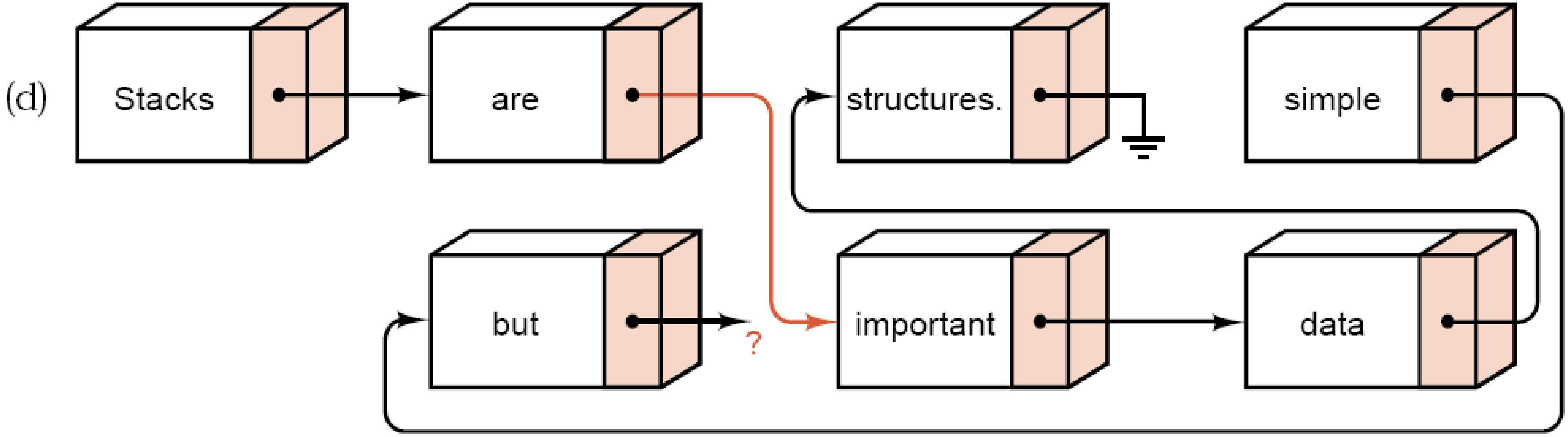
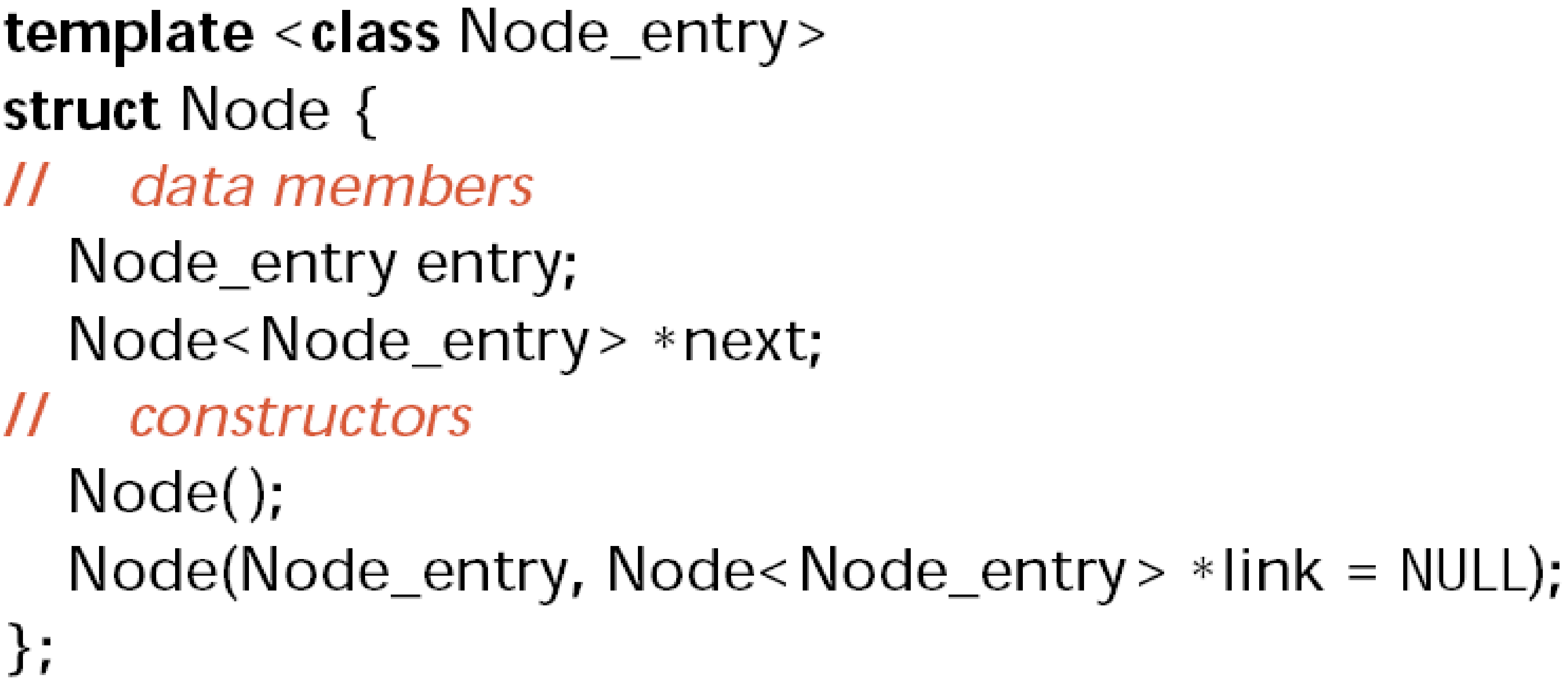


Figure 6.1. Actions on a linked list

6.2.3 Simply Linked Implementation



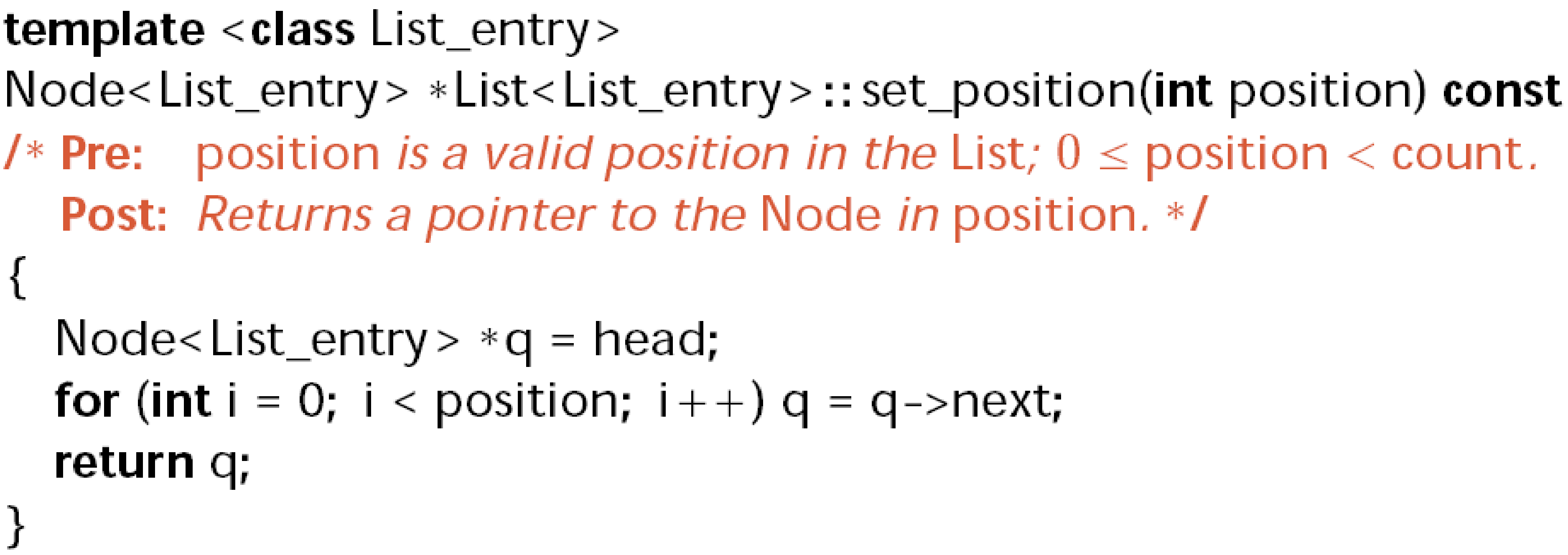
```
template <class Node_entry>
struct Node {
//    data members
   Node_entry entry;
   Node<Node_entry> *next;
//    constructors
   Node( );
   Node(Node_entry, Node<Node_entry> *link = NULL);
};
```

6.2.3 Simply Linked Implementation

```
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;
};
```

6.2.3 Simply Linked Implementation



```
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;
}
```

6.2.3 Simply Linked Implementation

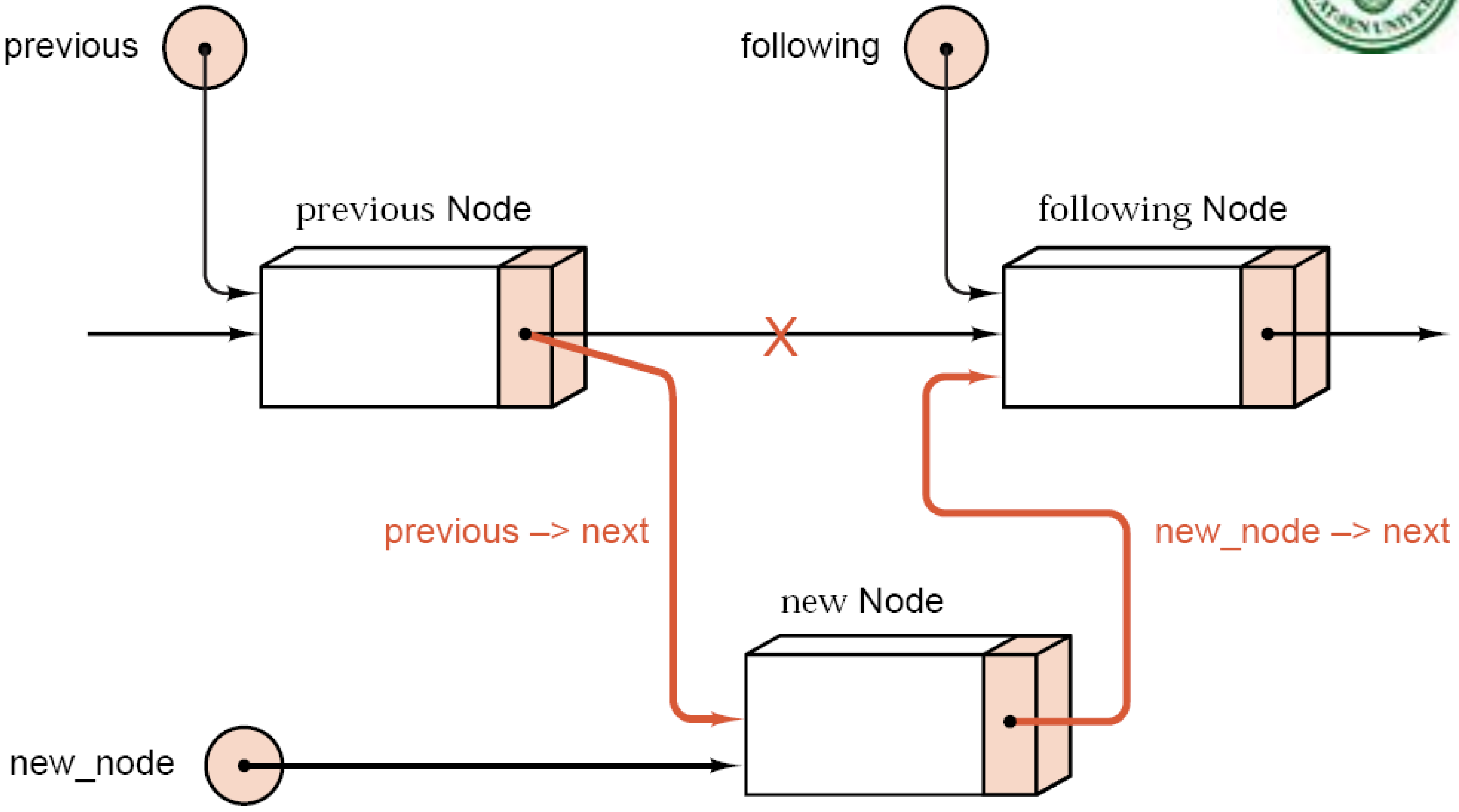
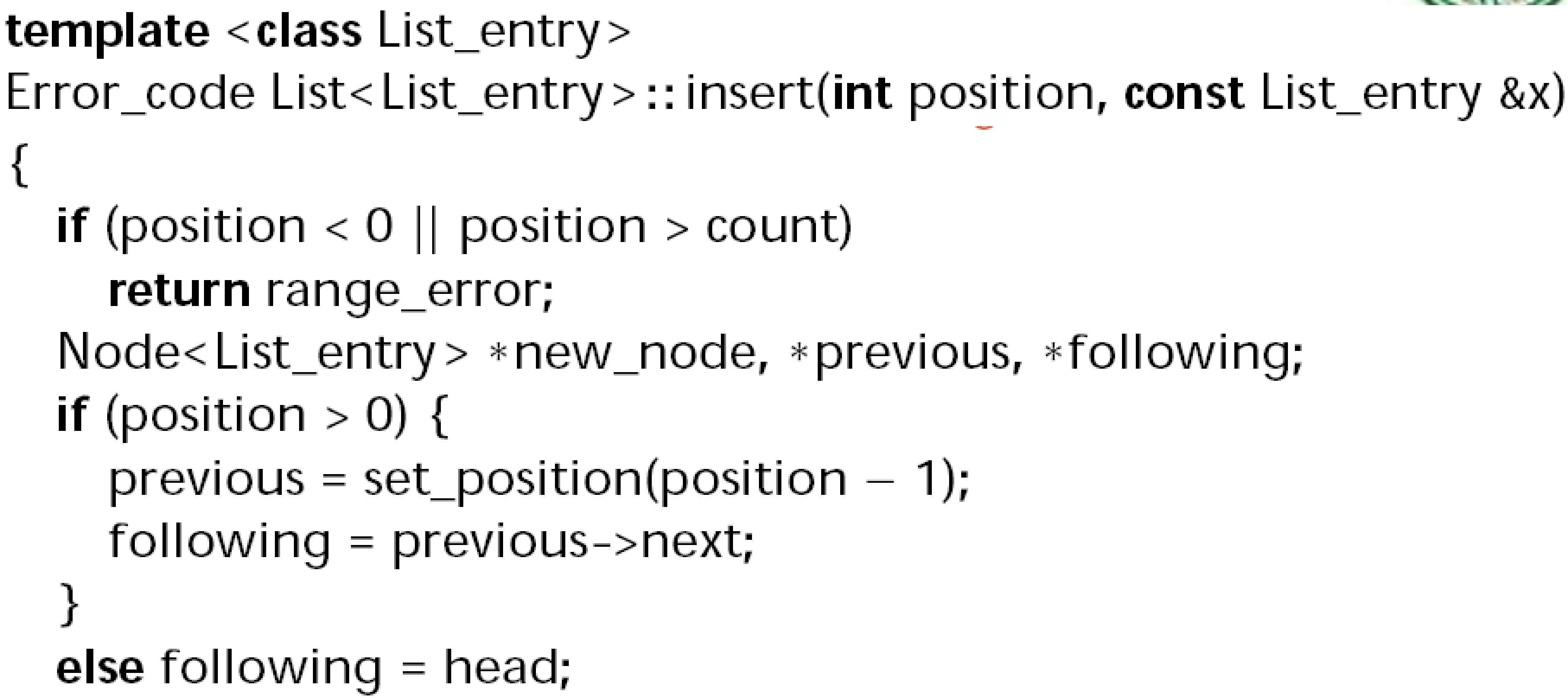


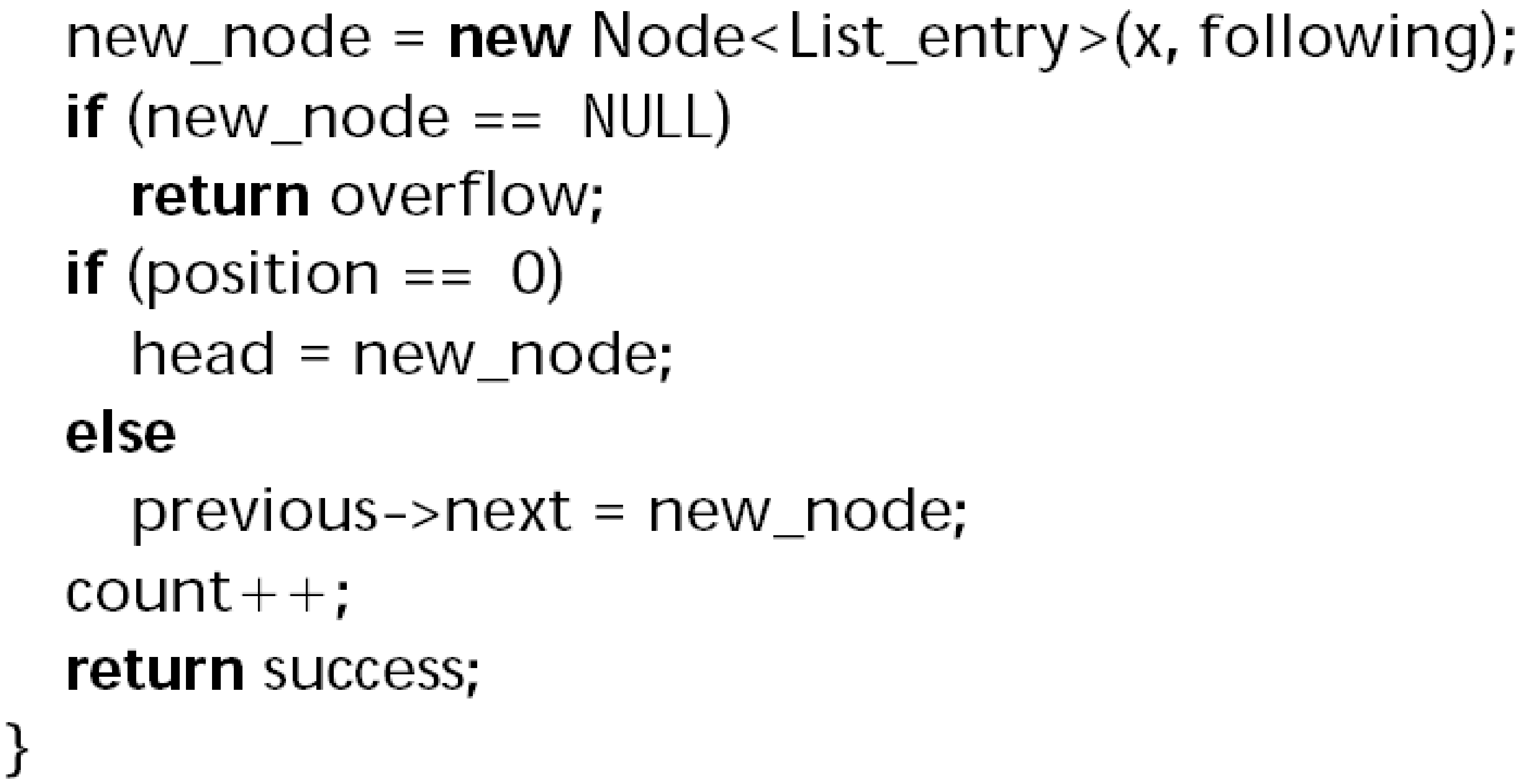
Figure 6.2. Insertion into a linked list

6.2.3 Simply Linked Implementation



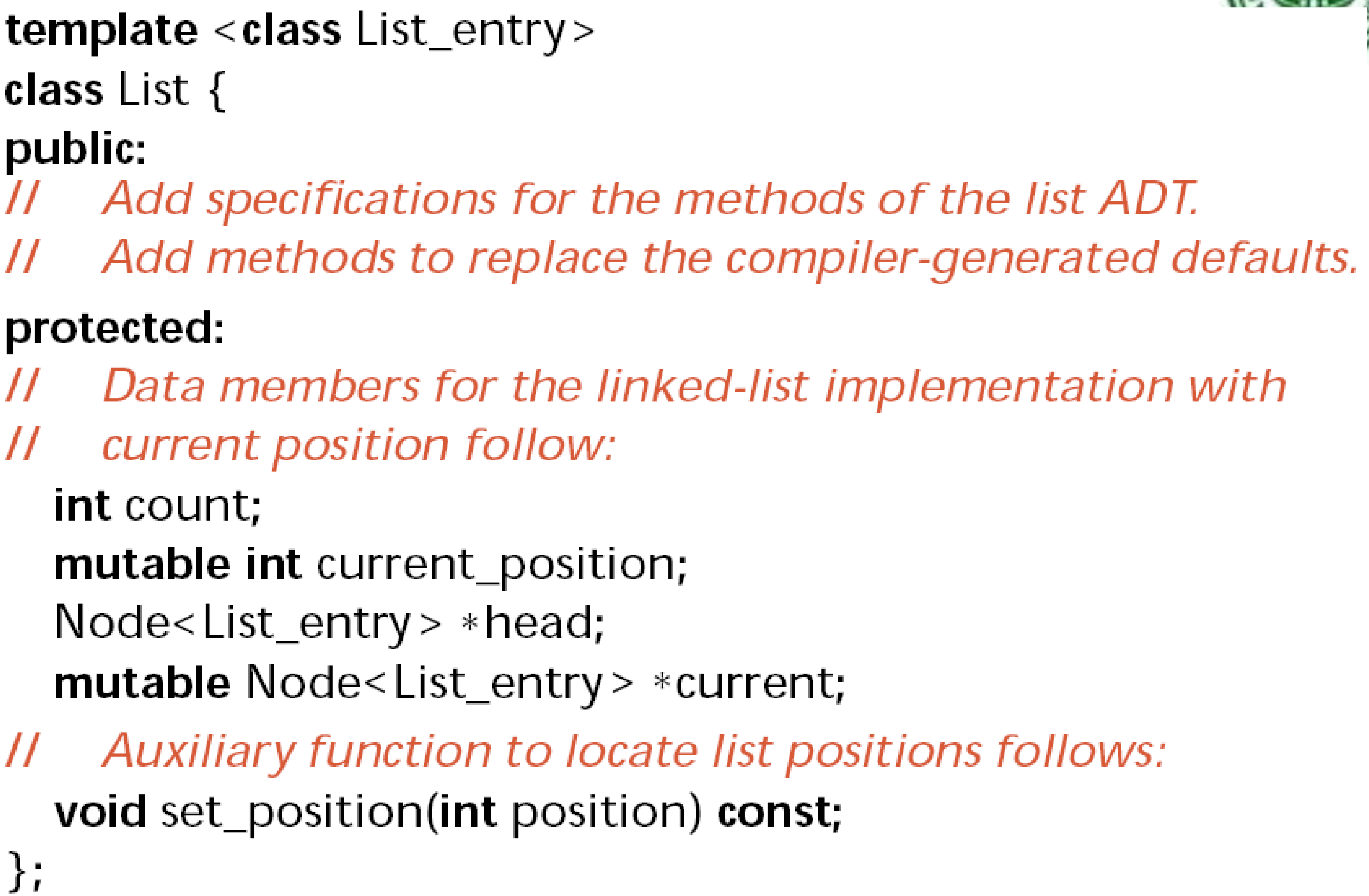
```
template <class List_entry>
Error_code List<List_entry>::insert(int position, const List_entry &x)
{
   if (position < 0 || position > count)
      return range_error;
   Node<List_entry> *new_node, *previous, *following;
   if (position > 0) {
      previous = set_position(position − 1);
      following = previous->next;
   }
   else following = head;
```

6.2.3 Simply Linked Implementation



```
   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;
}
```

6.2.4 Variation: Keeping the Current 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;
};
```

6.2.4 Variation: Keeping the Current 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;
}
```

6.2.5 Doubly Linked Lists

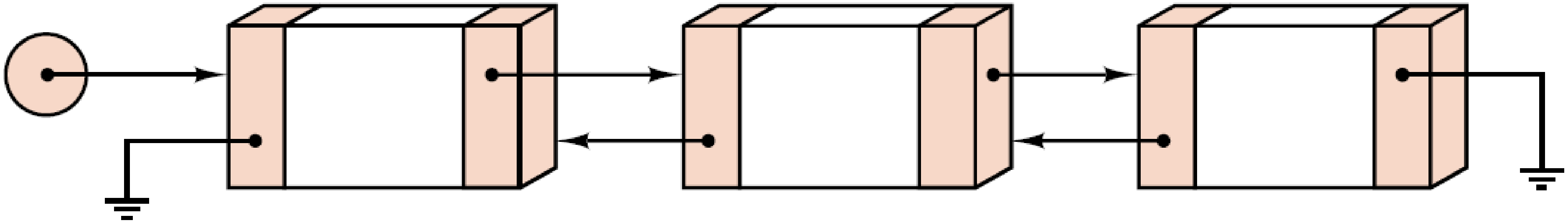
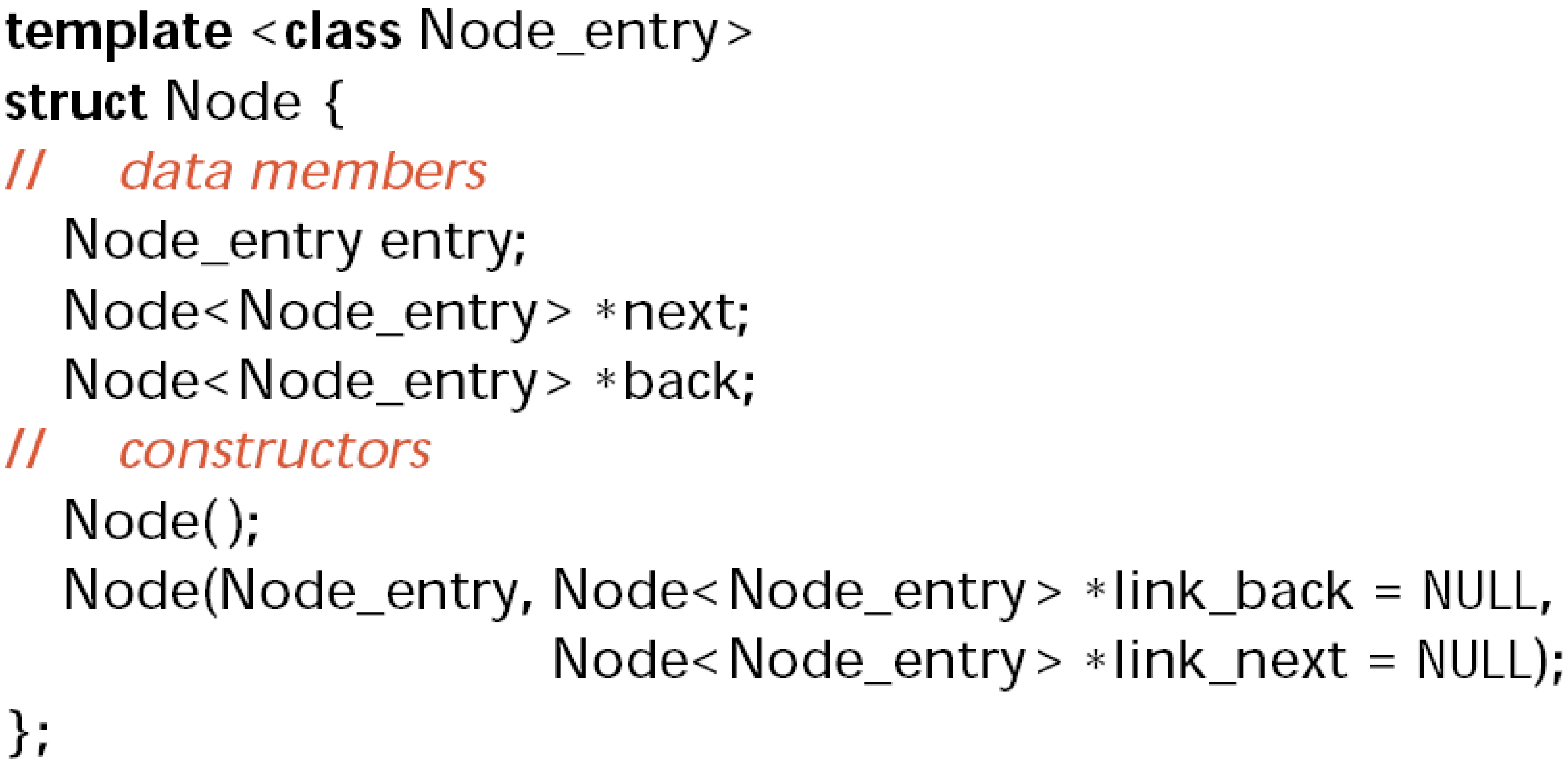


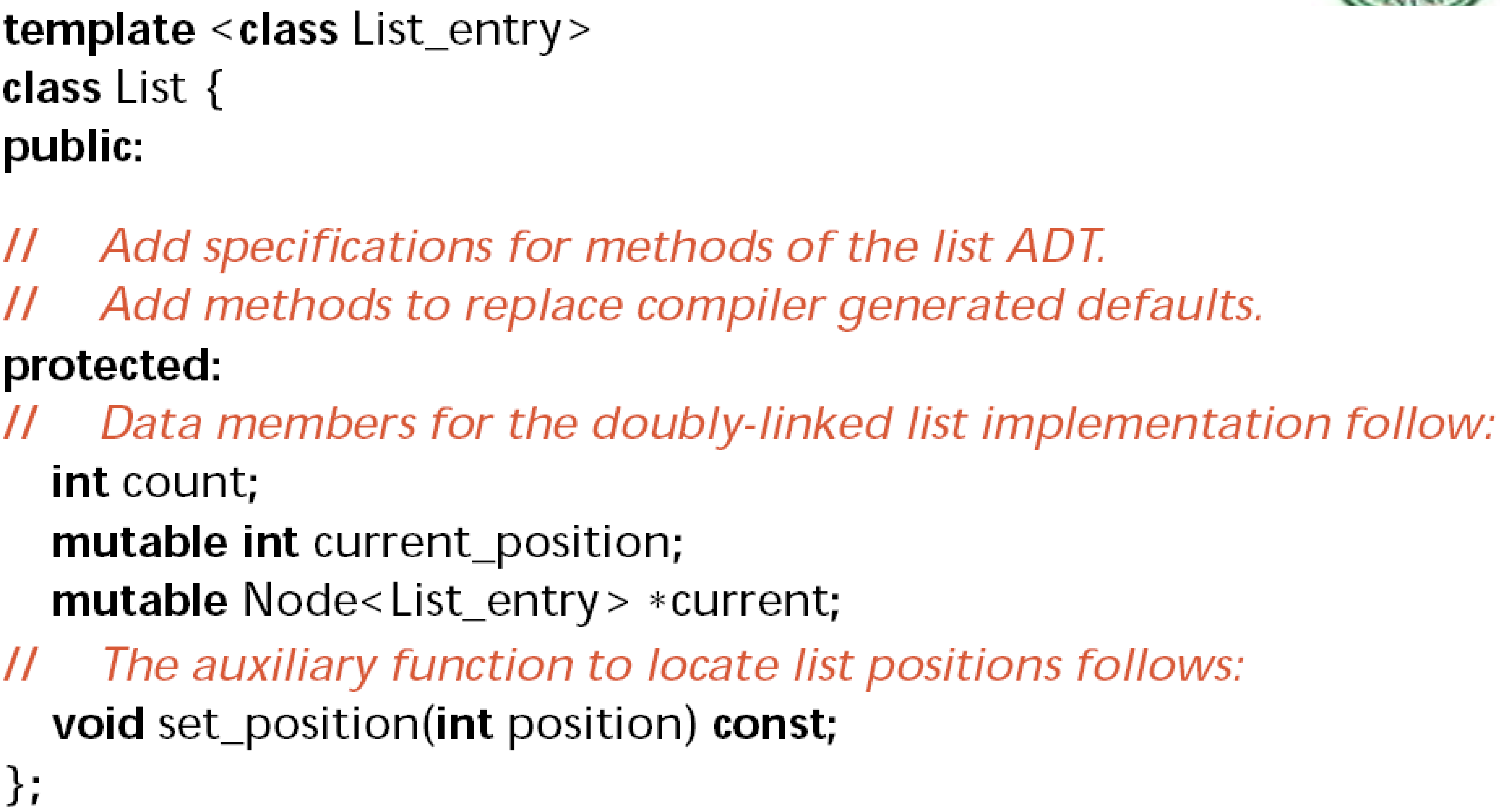
Figure 6.3. A doubly linked list

6.2.5 Doubly Linked Lists



```
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);
};
```

6.2.5 Doubly Linked Lists



```
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;
};
```

6.2.5 Doubly Linked Lists

```
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;
}
```

6.2.5 Doubly Linked Lists

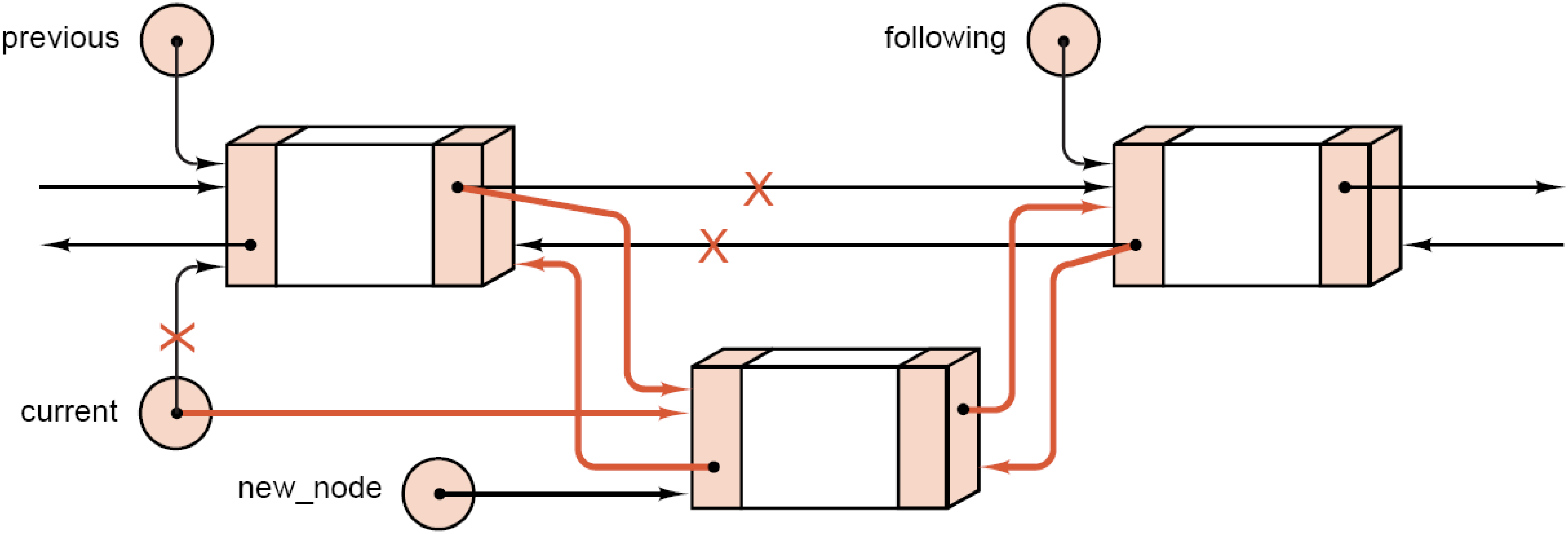


Figure 6.4. Insertion into a doubly linked list

6.3.1 Strings

- C-strings（char *)
- String

6.3.1 Strings in C++

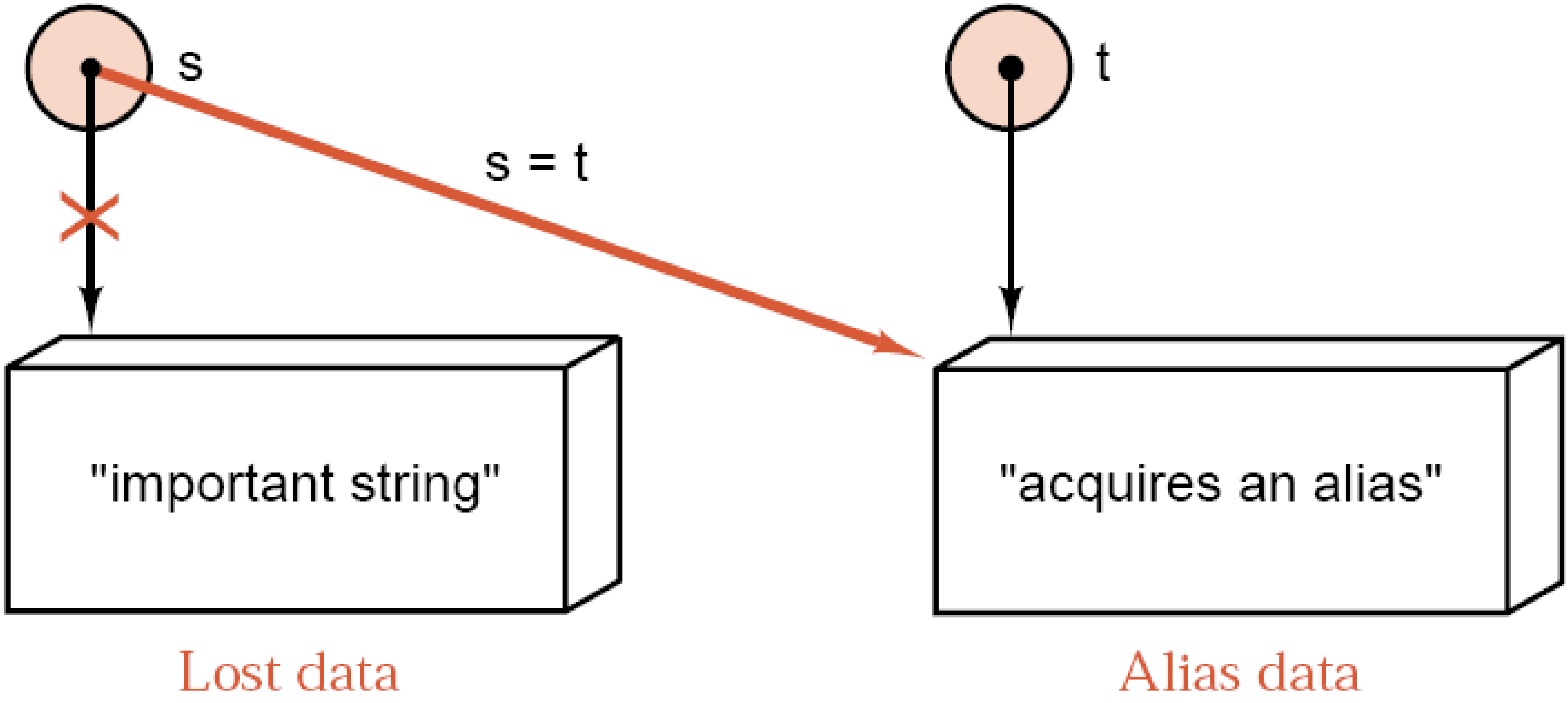
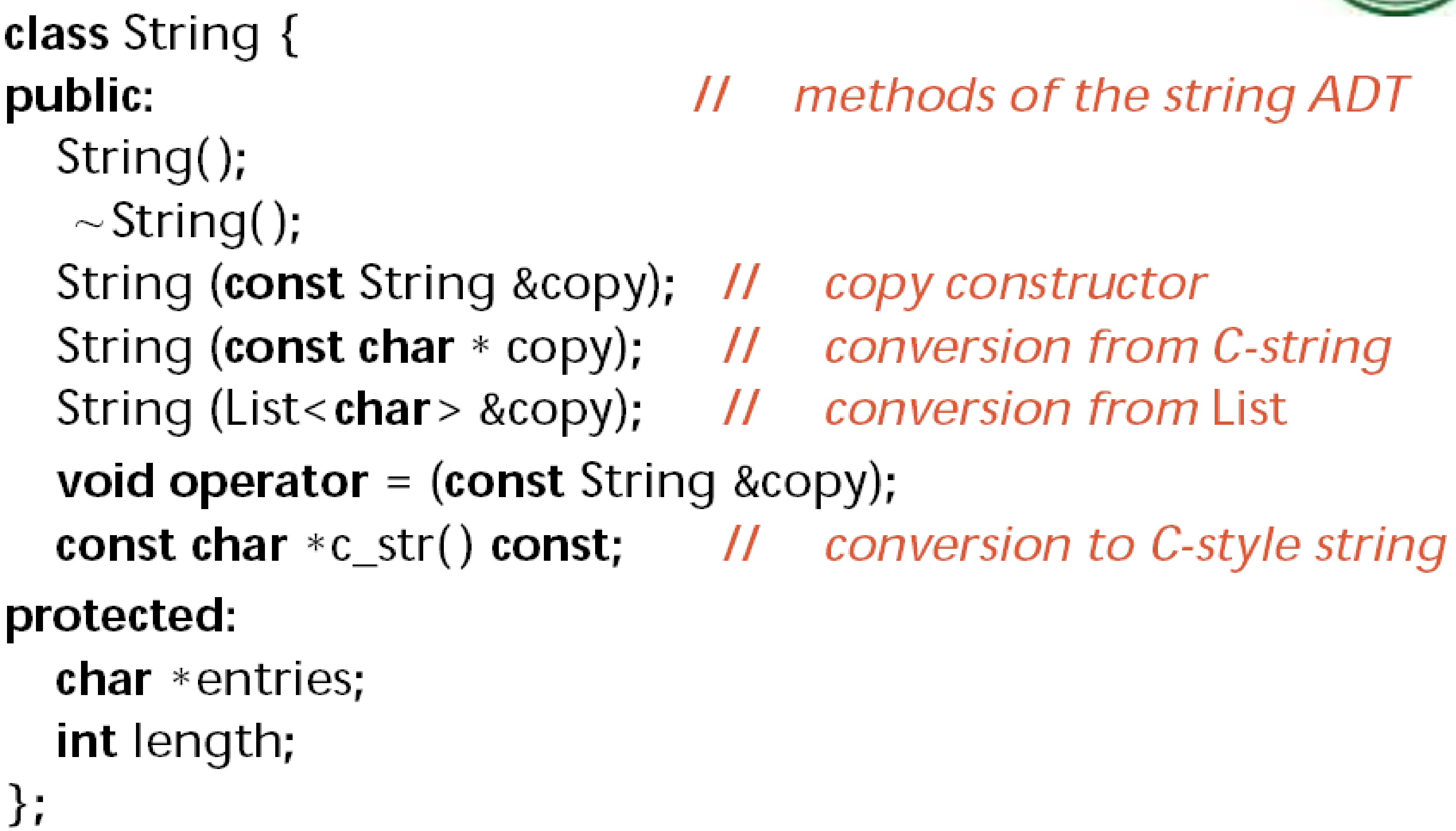


Figure 6.5. Insecurities of C-string objects

6.3.2 Implementation of Strings



```
class String {
public:                               //    methods of the string ADT
   String();
   ~String();
   String (const String &copy);       //    copy constructor
   String (const char * copy);        //    conversion from C-string
   String (List<char> &copy);         //    conversion from List
   void operator = (const String &copy);
   const char *c_str() const;         //    conversion to C-style string
protected:
   char *entries;
   int length;
};
```

6.3.2 Implementation of Strings

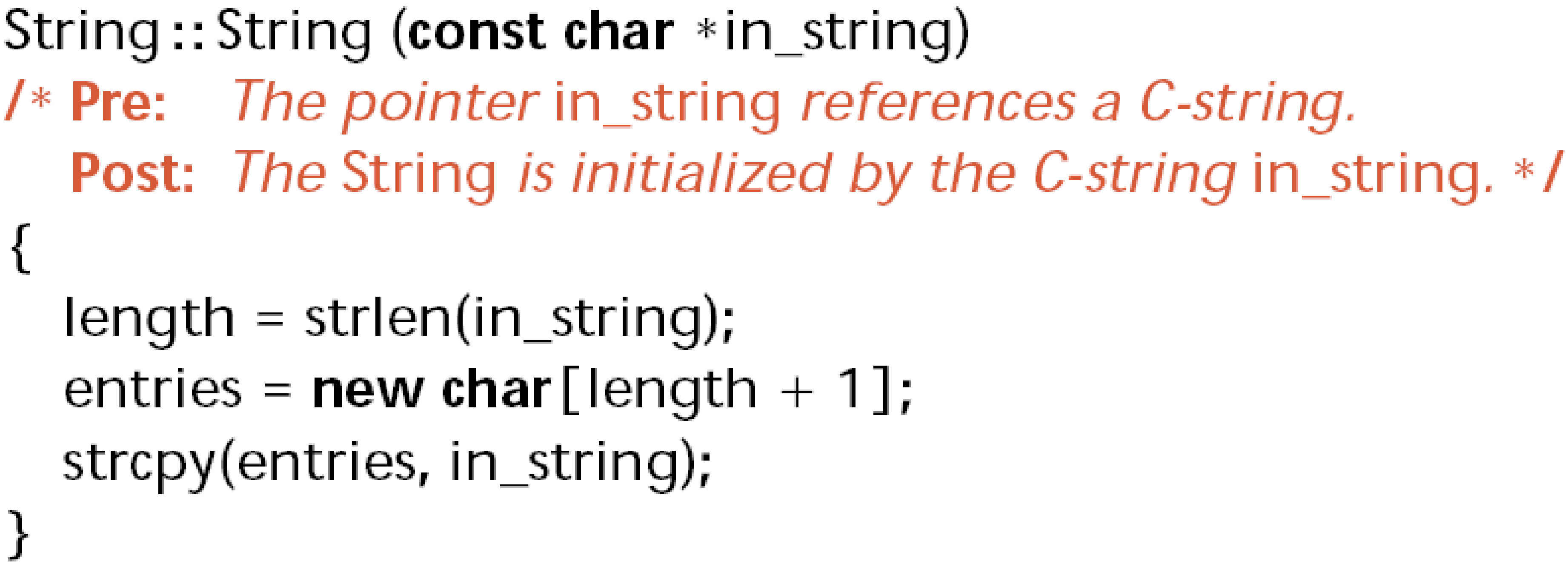
```
String::String()
//Post: A new empty String object is created.
{
   length = 0;
   entries = new char[length + 1];
   strcpy(entries, "");
}

String::~String()
//Post: The dynamically acquired storage of a String is deleted.
{
   delete []entries;
}
```

6.3.2 Implementation of Strings

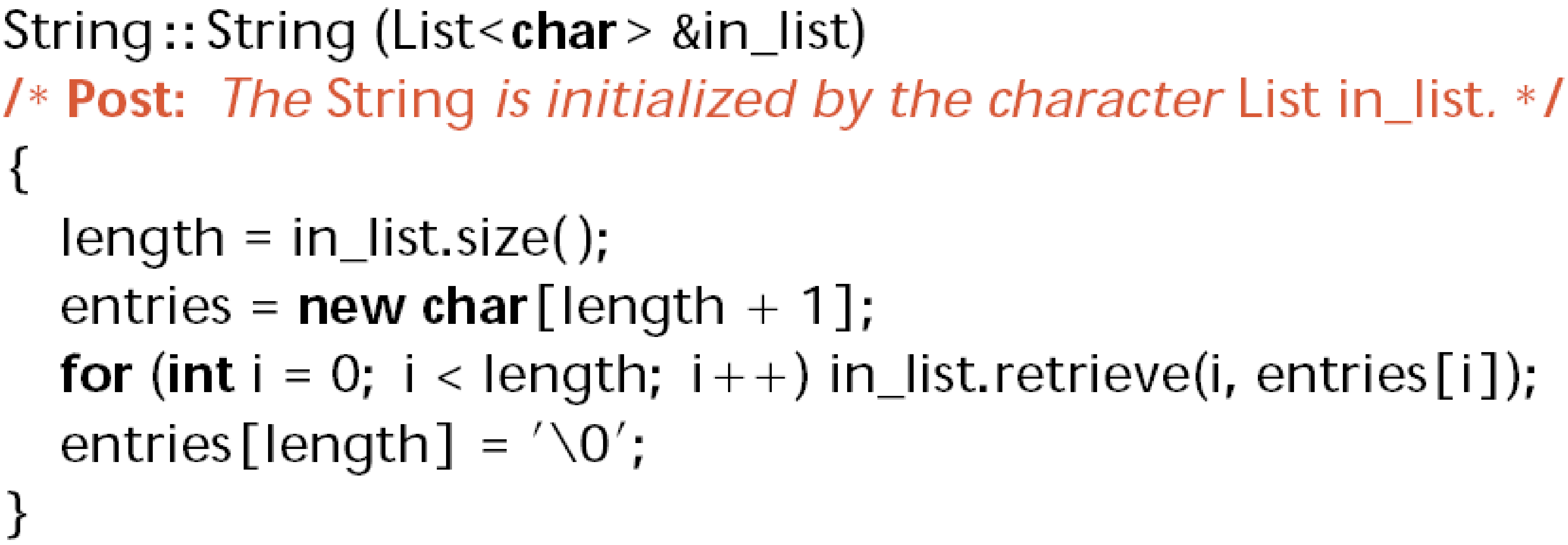
```
String::String(const String &copy)
//Post: A new String object is created to match copy.
{
   length = strlen(copy.entries);
   entries = new char[length + 1];
   strcpy(entries, copy.entries);
}
```

6.3.2 Implementation of Strings



```
String :: String (const char *in_string)
/* Pre:  The pointer in_string references a C-string.
   Post: The String is initialized by the C-string in_string. */
{
   length = strlen(in_string);
   entries = new char[length + 1];
   strcpy(entries, in_string);
}
```

6.3.2 Implementation of Strings



```
String :: String (List<char> &in_list)
/* Post: The String is initialized by the character List in_list. */
{
   length = in_list.size();
   entries = new char[length + 1];
   for (int i = 0; i < length; i++) in_list.retrieve(i, entries[i]);
   entries[length] = '\0';
}
```

6.3.2 Implementation of Strings

```
void String:: operator =(const String &copy)
//Post: A String object is assigned the value of the String copy.
{
   if (strcmp(entries, copy.entries) != 0)
   {
      delete []entries;
      length = strlen(copy.entries);
      entries = new char[length + 1];
      strcpy(entries, copy.entries);
   }
}
```

6.3.2 Implementation of Strings

```
const char*String::c_str() const
{
   return (const char *) entries;
}
```

e.g.,

```
string s = "abc";
const char *new_string = s.c_str();
s = "def";
cout << new_string;
```

6.3.2 Implementation of Strings

```
bool operator ==(const String &first, const String &second);
bool operator  >(const String &first, const String &second);
bool operator  <(const String &first, const String &second);
bool operator >=(const String &first, const String &second);
bool operator <=(const String &first, const String &second);
bool operator !=(const String &first, const String &second);

bool operator ==(const String &first, const String &second)
/* Post: Return true if the String first agrees with
String second.  Else: Return false.*/
{
   return strcmp(first.c_str(), second.c_str()) == 0;
}
```

6.3.3 Further String Operations

```
void strcat(String &add_to, const String &add_on)
/* Post: The function concatenates String add_on
onto the end of String add_to.*/
{
   const char *cfirst = add_to.c_str();
   const char *csecond = add_on.c_str();
   char *copy = new char[strlen(cfirst) + strlen(csecond) + 1];
   strcpy(copy, cfirst);
   strcat(copy, csecond);
   add_to = copy;
   delete []copy;
}
```

6.3.3 Further String Operations

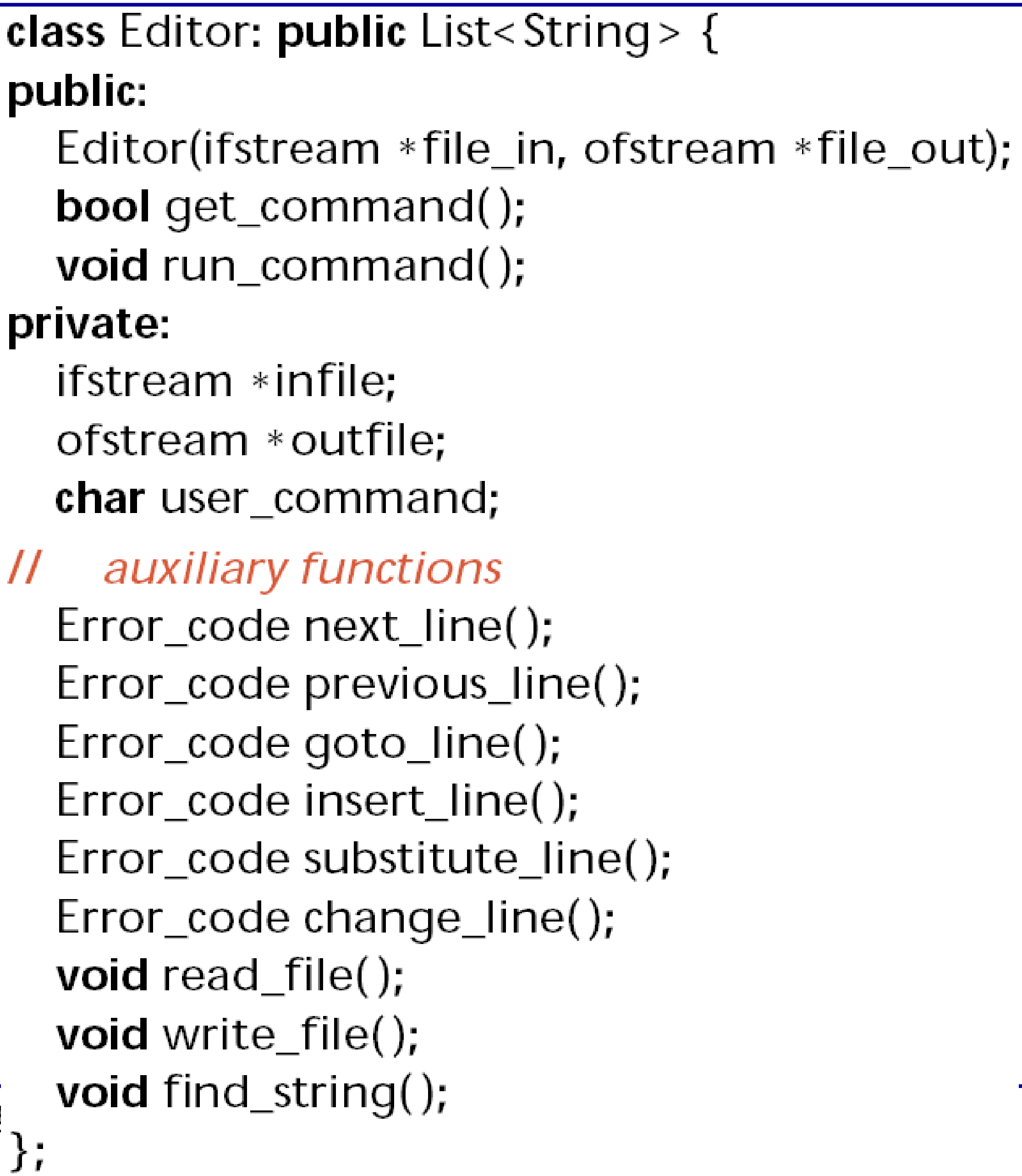
```
String read_in(istream &input)
/* Post: Return a String read (as characters terminated
by a newline or an end-of-file character)
from an istream parameter.*/
{
   List<char> temp;
   int size = 0;

   char c;
   while ((c = input.peek()) != EOF && (c = input.get()) != '\n')
      temp.insert(size++, c);
   String answer(temp);
   return answer;
}
```

6.4 A Text Editor

- We shall consider each line of text in an Editor object to be a string.
- The Editor class will be based on a List of strings.

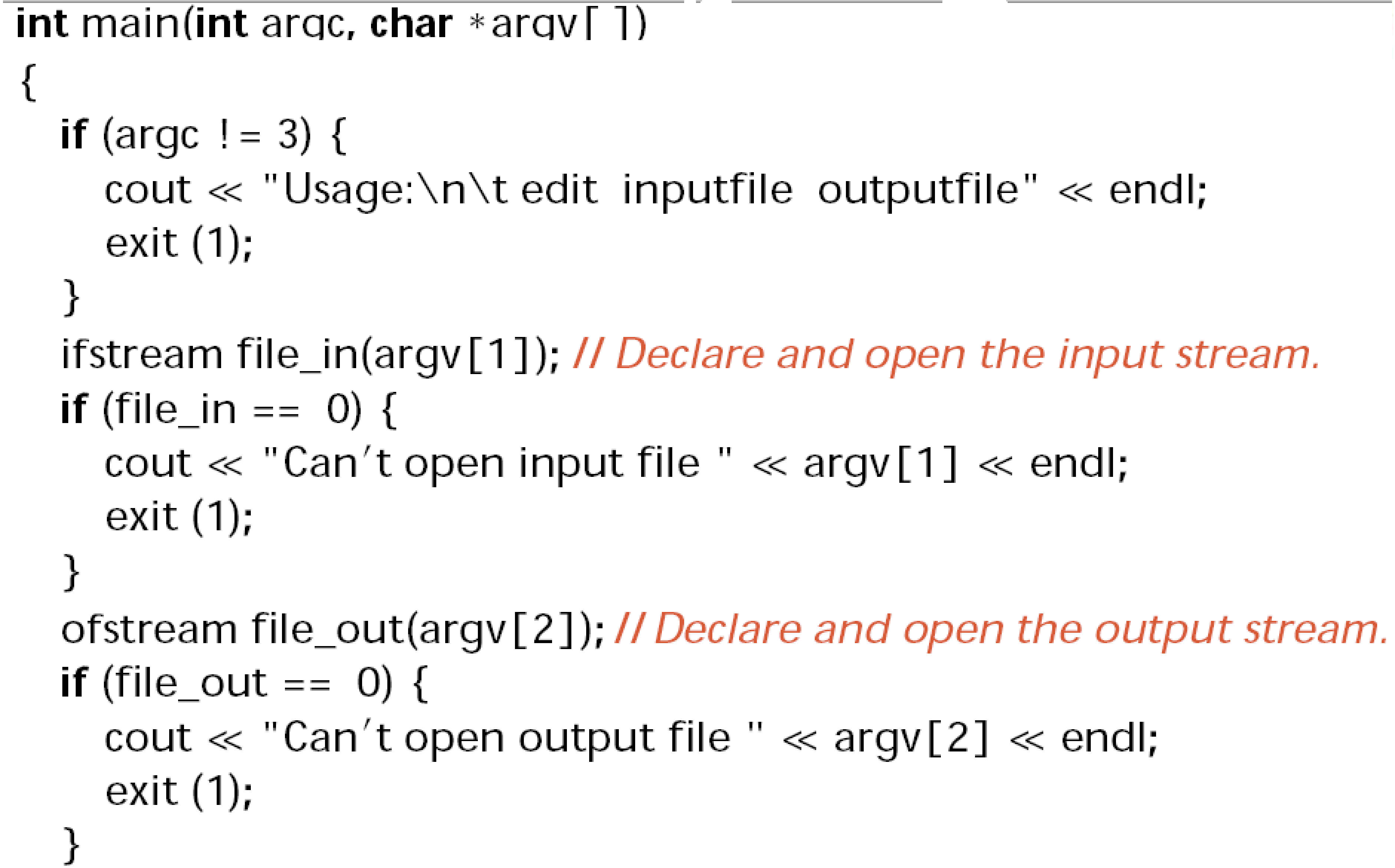
6.4.2 Implementation



```
class Editor: public List<String> {
public:
   Editor(ifstream *file_in, ofstream *file_out);
   bool get_command( );
   void run_command( );
private:
   ifstream *infile;
   ofstream *outfile;
   char user_command;

//   auxiliary functions
   Error_code next_line( );
   Error_code previous_line( );
   Error_code goto_line( );
   Error_code insert_line( );
   Error_code substitute_line( );
   Error_code change_line( );
   void read_file( );
   void write_file( );
   void find_string( );
};
```

6.4.2 Implementation

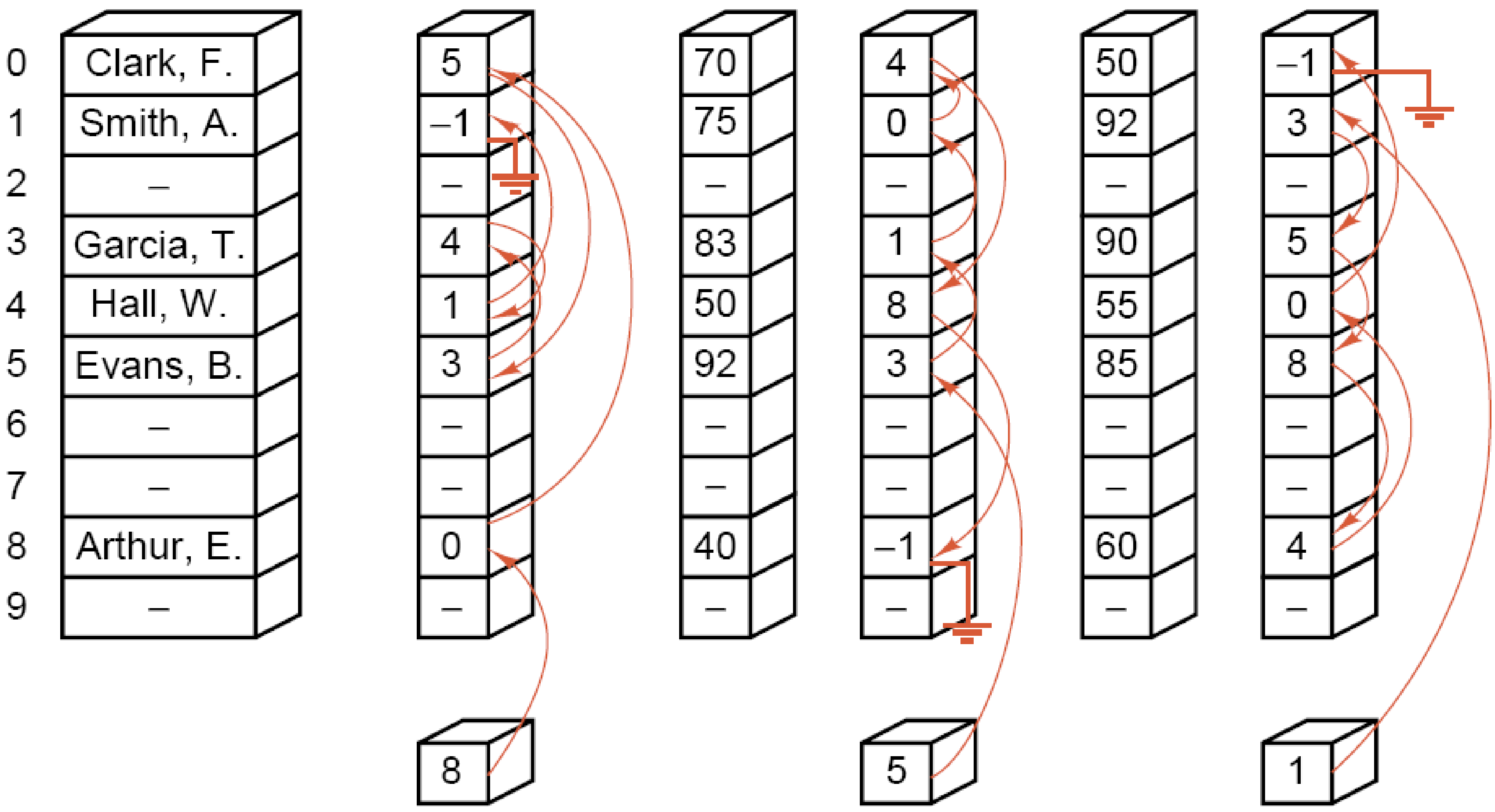


```
int main(int argc, char *argv[ ])
{
   if (argc != 3) {
      cout ≪ "Usage:\n\t edit  inputfile  outputfile" ≪ endl;
      exit (1);
   }
   ifstream file_in(argv[1]); // Declare and open the input stream.
   if (file_in ==  0) {
      cout ≪ "Can′t open input file " ≪ argv[1] ≪ endl;
      exit (1);
   }
   ofstream file_out(argv[2]); // Declare and open the output stream.
   if (file_out ==  0) {
      cout ≪ "Can′t open output file " ≪ argv[2] ≪ endl;
      exit (1);
   }
```

6.4.2 Implementation

```
bool Editor::get_command()
{
   if (current != NULL)
      cout << current_position << " : "
           << current->entry.c_str() << "\n??" << flush;
   else
      cout << "File is empty. \n??" << flush;
   cin >> user_command;    //  ignores white space and gets
command
   user_command = tolower(user_command);
   while (cin.get() != '\n')
      ;                    //  ignore user's enter key
   if (user_command == 'q')
      return false;
   else
      return true;
}
```

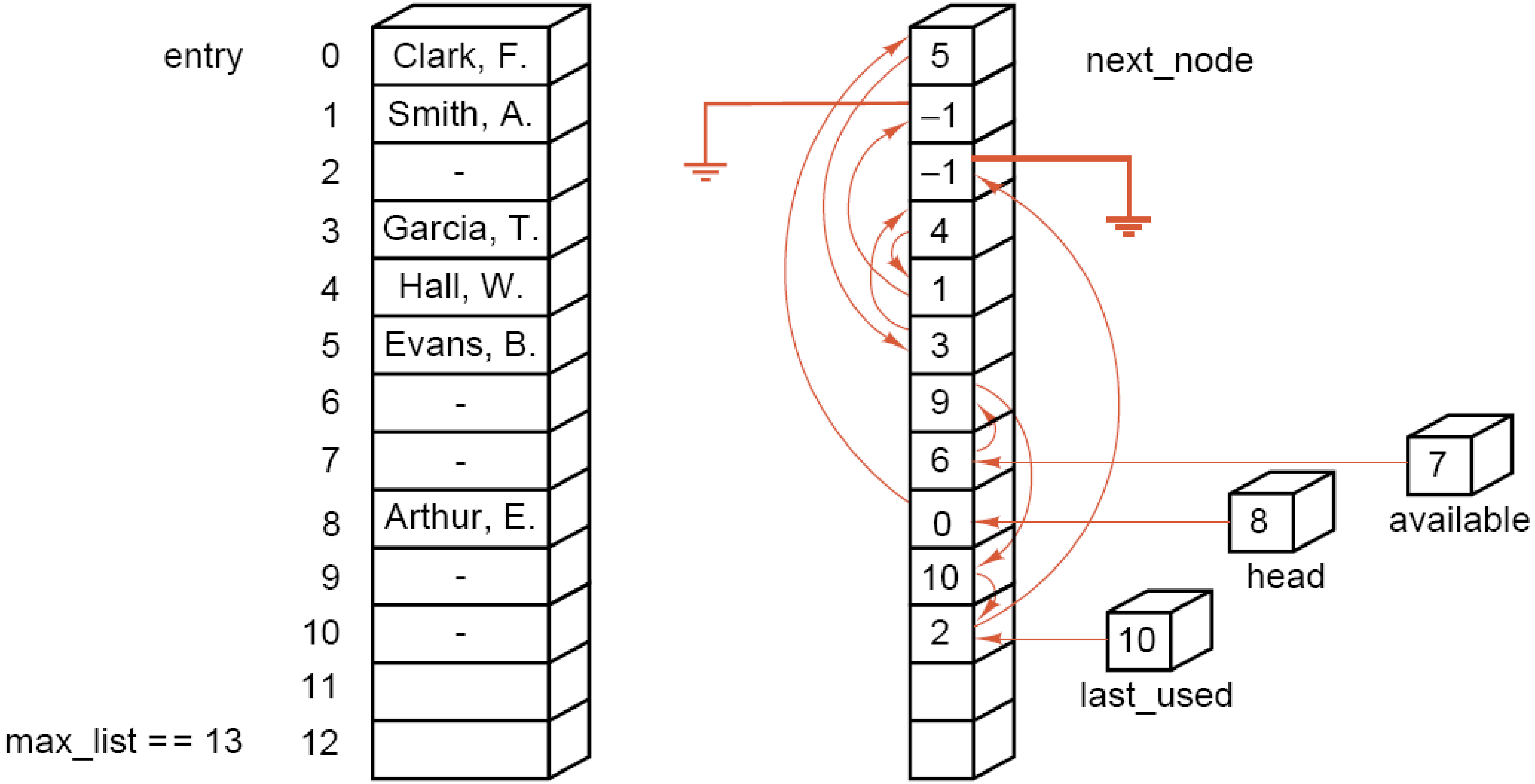
6.5 Linked Lists in Arrays



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

Figure 6.6. Linked lists in arrays

6.5 Linked Lists in Arrays



	entry	next_node
0	Clark, F.	5
1	Smith, A.	–1
2	-	–1
3	Garcia, T.	4
4	Hall, W.	1
5	Evans, B.	3
6	-	9
7	-	6
8	Arthur, E.	0
9	-	10
10	-	2
11		
12		

Figure 6.7. The array and stack of available space

6.6 Application: Generating Permutations

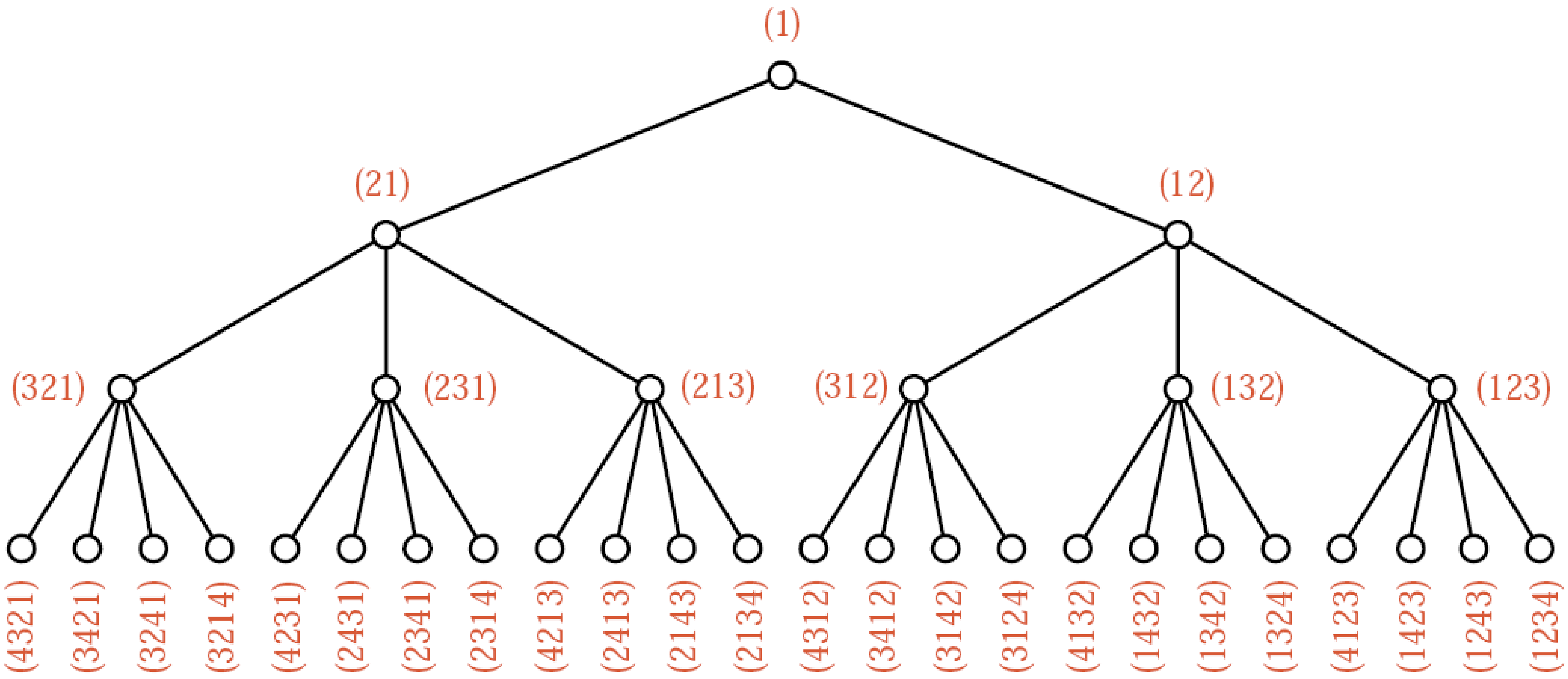


Figure 6.8. Permutation generation by multiplication, $n = 4$