In Chapter 4 we used linked structures to maintain stacks and queues. The principal advantage they enjoy over arrays for implementing those data types is that they require no \textit{a priori} estimate of the maximum size to which the data structure will grow. In this chapter we shall use pointers to link together dynamically allocated structures in more flexible ways than we saw in Chapter 4. The first two sections present the ideas for several important operations on linked list data structures and an application of linked lists to store sets. Later sections describe some tricks of the linked-list trade and some more tangled list structures.

5.1 LISTS

Suppose we had some items stored in structures whose \textit{info} member contains data and whose \textit{next} member contains a pointer to another such structure of the same type. Figure 5.1 shows how we could

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{linked_list_diagram}
\caption{A linked list.}
\label{fig:linked_list}
\end{figure}
In Algorithm 5.3

Algorithm 5.3

\[ \begin{align*}
\text{Find the first item that matches description } d \text{ in the linked list that starts at } \text{head}. \\
\text{Algorithm 6.1}
\end{align*} \]

\[ \begin{align*}
\text{function } p \leftarrow d \\
\text{while } (\text{not } p \text{ is null) do } \\
\text{next} \leftarrow \text{read} \\
\text{if } \text{description} p = \text{description} \text{next} \\
\text{then } \text{head }= \text{next} \\
\text{return } \text{next}
\end{align*} \]

Inserting an Element

The insertion of a new element into a linked list is a common operation. When inserting a new element, we need to find the position where the new element should be placed. Once the position is determined, the new element is inserted into the list at that position. This operation is typically performed using the function `insert(element, position)`, which takes an element and a position as input and inserts the element at the specified position in the list.

Advantages over Arrays

One advantage of using linked lists over arrays is that they do not impose an upper bound on the size of the list. This can be especially useful in applications where the size of the list is not known in advance or is expected to change frequently. Additionally, inserting and deleting elements in a linked list can be done in constant time, making it more efficient than arrays in certain cases.

Disadvantages over Arrays

One disadvantage of using linked lists over arrays is that they require additional memory for the pointers. Each element in a linked list contains a pointer to the next element, which can lead to increased memory usage compared to arrays. Additionally, accessing elements in a linked list requires traversing the list from the beginning, which can be slower than accessing elements in an array, especially if the element is not at the beginning or end of the list.

Finding the Item

To find an item in a linked list, we can traverse the list from the beginning until we find the item. The time complexity of this operation is \( O(n) \) in the worst case, where \( n \) is the number of items in the list. In the best case, if the item is at the beginning of the list, the time complexity is \( O(1) \).

Inserting an Element

To insert a new element into a linked list, we first need to find the position where the new element should be placed. This can be done using the `find(element)` function, which returns the position of the given element in the list. Once the position is found, we can insert the new element at that position using the `insert(element, position)` function.

Algorithm for Insertion

\[ \begin{align*}
\text{function } \text{insert(element, position)} \\
\text{if position is valid and not out of bounds} \\
\text{then } \text{element} \leftarrow \text{list[position]} \\
\text{list[position]} = \text{element} \\
\text{return } \text{list}
\end{align*} \]
Algorithm 5.4

Local situation before deletion of item doomed.

Figure 5.4

The situation after new has been inserted after p. The rest of the data structure.

Figure 5.5

Since we can insert an item after a given item in a linked list in con-

stant time, we might hope to perform a similar feat when deleting an

array, we must shift some of the remaining items by one place in the

array before new is inserted after p. The unutilized array points at

and p->next into the data structure.

p and its arrow that points at p->next indicate the pointers that point

to the item before new is inserted after p. The next element after

the insertion before new is inserted after p. The rest of the data structure.

Algorithm 5.3

d->next = new
new->next = d->next
item->prev->next (new)
To illustrate programming with linked lists, we shall use them to store finite sets of elements from an ordered universe. These "sets" are the garden-variety sets of elementary mathematics: they contain elements from a universe, and no element appears more than once in a set; hence the term "linked list" will denote a sequence of different sets of elements from a universe. Since an implementation will require different sets of operations, we shall provide these operations in a consistent manner, where applicable.

We begin with Program 5.1, a diagnostic function that prints the contents of the set.

**Figure 5.6**

```
| insert(s, e) if e is not already present |
| delete(s, e) if e is in set s |
| members(s) if s is empty |
| empty(s) if s is empty |
```

**Figure 6.5**

Local situation after item 'd' has been deleted.
A function to test whether an item belongs to a set.

PROGRAM 6's

```c
{ 
    return s == value ?
        s
    : s->next;
/* ASSERT: n has not been seen yet */
}

while (s->value > n)
    s = s->next;

if (s->value == n) return true;
else return false;

boolean member(s, n)
```

Inserting a New Element

When we write `insert(s, x)`, we must decide first what type of

the data structure is to build and define the program.

structure of a program. We can use `dumpset()` to take snapshots of

A function to print out the contents of a set.

PROGRAM 7's

```c
{ 
    printf("\n");
    { 
        s = s->next;
        printf("%,d\n", s->value);
    } while (s->value > n)
```

Searching

When we begin to write member(s, n), it is tempting to draw a

structure definitions for sets stored as ordered linked lists.

PROGRAM 8.4

```c
{ 
    while (s->value > n)
        s = s->next;
```

```c
typedef struct statement *set;
```
In all cases, points to the node after which new should be inserted.

If the pointer is at the last element

- if the new element's position is equal to the last element of the list,
  then the last element becomes the first element of the list.
- if the new element's position is after the list's last element, then
  the last element is null, so $s->next$ is null.

boundary case above!

Let's see how this solution fares in the face of the troublesome

```c
while (p->next && p->next->value < u)
    p = p->next;
```

or what should be above?

The following loop leaves $s$ pointing to the element after which

```c
while (s->next && s->next->value > u)
    s = s->next;
```

would be inserted (or $s$ is larger than any element of the list); it

- should leave $s$ pointing to the element before which $u$ should
- have been inserted.

This problem is member (u) does not find $u$ in the list.

```c
    if (s->next == NULL)
        return p;
    while (s->next && s->next->value < u)
        s = s->next;
    if (s->next && s->next->value == u)
        return s->next;
    s = s->next;
    while (s->next && s->next->value > u)
        s = s->next;
    return s;
```

An element insertion algorithm that uses a tailing pointer $s$ that follows along

```c
#include <stdio.h>

int main()
{
    int a[] = {1, 2, 3, 4, 5};
    int i, n = 5;
    printf("Enter the value to insert: ");
    scanf("%d", &x);
    if (x < a[0])
        insert(a, n, x, 0);
    else if (x > a[n-1])
        insert(a, n, x, n - 1);
    else
    {
        int j = 0;
        while (a[j] < x)
            j++;
        insert(a, n, x, j);
    }
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
    return 0;
}
```

A function to create a space for an item of a set

```c
PROGRAM 6.12
{
    return;
}
```

```c
extern int = m, n;
char *string = "Hello World";
```
Program 5.7 shows another approach to element insertion. The idea is to use a linked list of dummy nodes that contain no elements, to serve as the head of the list, and to insert new elements into this list. When a new element is inserted, it is added to the list as a new node, and the reference to the head of the list is updated to point to the new node. This approach is similar to the one used in the previous program, but it has the advantage that it does not require any special handling of the head of the list.

ALGORITHM 5.6

```
// Function to insert a new element into the list.

program insert(list); // dummy node

{ return dummy.next; } 

if (new.next = d.next) 

    new = new.next; // move new to the end

else

    set insert(new, d); // insert new

```

The algorithm for inserting a new element into the list is as follows:

1. If the new element is equal to the last element of the list, append it to the end.
2. Otherwise, find the last element of the list and insert the new element as a new node after it.

The use of a dummy node at the head of the list is a convenient way to simplify the handling of the head of the list, and it makes it easier to insert elements into the list.

Because the function can change the value of the list head.

```plaintext
s = insert(s, n);
```

In sets, the correct way to call `insert()` is to preserve the data structure invariant that copies duplicate elements.

```
function insert(element):
    // Insert element into the set.
```

Figure 5.7: Illustration of another approach to element insertion. The idea is to use a linked list of dummy nodes that contain no elements, to serve as the head of the list, and to insert new elements into this list. When a new element is inserted, it is added to the list as a new node, and the reference to the head of the list is updated to point to the new node.
Laemmer's algorithm to form the union of two sets stored as ordered linked lists

PROGRAM 5.17

```c

{ /*

for each element in first set
    if element not in second set
        add element to second set

*/

```
Header Nodes

Each can be applied separately or in conjunction with the others.

In this section we shall see several techniques that can be used in

Michellaneous tools for Linked Structures

5.3

3.5.3. Miscellaneous tools for Linked Structures
Singly linked lists whose tail nodes point to their heads, as in Figure 5.11, then we could dispense with the first test in the
next-d = d
while (d ≠ next-value)
  * next = next.next
  * assume last element in list is end of list after this operation
  \nNote that it is already present in list then, but it's
the end
\nIn the list num even elements, we have to be sure not to go
beyond p.next.next, so we can omit the
first test altogether:

Next-d = d
while (d ≠ next-value)
  * next = next.next
  * assume last element in list is end of list after this operation
  \nField the second test in the
next-d = d
while (d ≠ next-value)
  * next = next.next
  * assume last element in list is end of list after this operation
  \nOn second example of sentence can be used even on an unordered
insert in an element must check to avoid running off the end of the list.

In the list:

This ordered linked list contains a sentinel value in its last node.

Figure 5.11

6.3 Miscellaneous Tools for Linked Structures

Figure 6.10
Doubly Linked Lists

In a doubly linked list, each node contains two pointers, one to the next node and one to the previous node. This allows for efficient traversal in both directions.

5.14

A doubly linked list. The drawing omits the explicit "box" that contains the pointer in each node. Of course, the actual node template must still have room for the pointer.

5.13

A circularly linked list. The drawing omits the explicit "box" that contains the pointer.

5.12

Will loop forever on a circular list.

\[
\text{while } d \neq \text{NULL} \\
\text{do } \\
\quad \text{d = d.next} \\
\quad \text{d = d.proc} \\
\quad \text{d = d.head} \\
\text{do}
\]

6.3 Miscellaneous Tools for Linked Structures

Doubly Linked Lists

When correctly traversing the circular list to which p points:

- If we know that the list is not empty, then the following algorithm will work:

  \[
  \text{do} \\
  \quad \text{while } d \neq \text{NULL} \\
  \quad \text{d = d.next} \\
  \quad \text{d = d.proc} \\
  \quad \text{d = d.head} \\
  \text{do}
  \]

- The drawing omits the explicit "box" that contains the pointer in each node. Of course, the actual node template must still have room for the pointer.
MULTIPLY LINKED STRUCTURES

5.4 MULTIPLY LINKED LISTS

5.4.1 MULTIPLY LINKED LISTS

In order to represent the columns of a matrix as a single two-dimensional array, we can use a doubly linked list. Each student can have a list of the classes they are enrolled in. Each student can also have a list of the classes in which they are enrolled in. If a student enrolls in a new class, their list of classes is updated accordingly. If a student drops a class, their list of classes is updated accordingly. The resulting structure is often called a multiply linked list.

In many applications, nodes have several pointers that link them together. These linked lists appear in many programs besides those of the procedure-se-

SUMMARY

Check that the doubly linked list is consistent. Each node during the traversal. On the way it uses a demand() to
Searches were unsuccessful. I thought a search function that would return the index of the element we’re looking for in the array was the right method to use if we wanted a search function that would return the index of the element we were looking for in an array, and we were wrong.

**Exercises**

Are you likely to be useful in the quest for consistently correct programs?

In the end, there is no single right way to build pointer-based programs. If we don't make sure that our programs are at least somewhat correct, it is easy to find an incorrect assertion that will make the program work. But even though the program is correct, one of the assertions that is true even though the program is correct is that the results of the program are correct. The results of the program are correct, but the program is incorrect.

If we write the results to the screen, we can see if the program is correct. But even though the program is correct, it is easy to find an incorrect assertion that will make the program work. But even though the program is correct, it is easy to find an incorrect assertion that will make the program work. But even though the program is correct, it is easy to find an incorrect assertion that will make the program work.

Figure 6.16

A pointer-based representation of a two-dimensional array.

Figure 6.17

A simple multithreaded structure.
2 What is the complexity of the functions in Program 5.17?

3 Revise function insert() to return a pointer to the new item if it is already there.

4 Implement a version of insert() that requires only one argument, the key.

5 Implement function insert(), of Program 5.18, so that the following function header is valid:

   void insert(s, n);

6 What difference does this make in how one calls insert()?

7 The version of insert() given in Algorithm 5.6 in terms of [s] and [n] for any [s] and [n] is not very intuitive. Discuss the merits of and problems with implementing a C function that precedes the smallest element to the front; this element in various data structures for sequences assuming that items are in the sequence.

8 Discuss the merits of inserting a new element to the front, middle, and rear of sequences assuming that items are in the sequence.

9 Complete this table of the worst-case time complexities of the various operations on various data structures for sequences assuming that items are in the sequence:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>Singly Linked List</th>
<th>Doubly Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty?</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Insert before</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Insert after</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Delete</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

10 Write a function that reverses the order in which the nodes appear on a singly linked list.

11 Write a function that can traverse a circularly doubly linked list in either direction.

12 Write a function that can traverse an ordered linked list in either direction.

13 Write a function that inserts a node after the sentinel node.

14 Write a function that could serve as the argument process to an application that inserts a new node into an ordered linked list.

15 An application that could serve as the argument process to an application that inserts a new node into an ordered linked list in either direction.

REFERENCES