Chapter 13

Pointers and Linked Lists
Overview

13.1 Nodes and Linked Lists
13.2 Stacks and Queues
13.1

Nodes and Linked Lists
Nodes and Linked Lists

- A linked list is a list that can grow and shrink while the program is running
- A linked list is constructed using pointers
- A linked list often consists of structs or classes that contain a pointer variable connecting them to other dynamic variables
- A linked list can be visualized as items, drawn as boxes, connected to other items by arrows
Nodes and Pointers

```
head

"rolls"
10

"jam"
3

"tea"
2
end marker
```
Nodes

The boxes in the previous drawing represent the nodes of a linked list.

- **Nodes contain the data item(s) and a pointer that can point to another node of the same type**
  - The pointers point to the entire node, not an individual item that might be in the node

- The arrows in the drawing represent pointers
Implementing Nodes

Nodes are implemented in C++ as structs or classes
- Example: A structure to store two data items and a pointer to another node of the same type, along with a type definition might be:

```c++
struct ListNode
{
    string item;
    int count;
    ListNode *link;
};
```

```c++
typedef ListNode* ListNodePtr;
```

This circular definition is allowed in C++
The head of a List

- The box labeled **head**, in display 13.1, is not a node, but a pointer variable that points to a node.

- Pointer variable head is declared as:

  ```
  ListNodePtr head;
  ```
Accessing Items in a Node

Using the diagram of 13.1, this is one way to change the number in the first node from 10 to 12:

\[(*\text{head}).\text{count} = 12;\]

- head is a pointer variable so *head is the node that head points to
- The parentheses are necessary because the dot operator . has higher precedence than the dereference operator *
The Arrow Operator

- The **arrow operator** -> combines the actions of the dereferencing operator * and the dot operator to specify a member of a struct or object pointed to by a pointer
  - \((\ast\text{head}).\text{count} = 12;\)
    - can be written as
    - head->count = 12;
  - The arrow operator is more commonly used
Accessing Node Data

head->count = 12;
head->item = "bagels";

Before

head

"rolls"
10
"jam"
3
"tea"
2
NULL

After

head

"bagels"
12
"jam"
3
"tea"
2
NULL
The defined constant `NULL` is used as…

- **An end marker for a linked list**
  - A program can step through a list of nodes by following the pointers, but when it finds a node containing `NULL`, it knows it has come to the end of the list

- The value of a pointer that has nothing to point to

The value of `NULL` is 0

Any pointer can be assigned the value `NULL`:

```c
double* there = NULL;
```
To Use NULL

- A definition of NULL is found in several libraries, including `<iostream>` and `<cstddef>`

- A using directive is not needed for NULL
Linked Lists

- The diagram in Display 13.2 depicts a linked list
- A linked list is a list of nodes in which each node has a member variable that is a pointer that points to the next node in the list
  - The **first node is called the head**
  - The pointer variable head, points to the first node
- The pointer named head is not the head of the list...it points to the head of the list
  - The **last node contains a pointer set to NULL**
Building a Linked List: The node definition

Let's begin with a simple node definition:

```c
struct Node {
    int data;
    Node *link;
};
```

typedef Node* NodePtr;
Building a Linked List: Declaring Pointer Variable head

With the node defined and a type definition to make or code easier to understand, we can declare the pointer variable head:

```c
NodePtr head;
```

- head is a pointer variable that will point to the head node when the node is created
To create the first node, the operator new is used to create a new dynamic variable:

```
head = new Node;
```

Now head points to the first, and only, node in the list.
Now that head points to a node, we need to give values to the member variables of the node:

```c
head->data = 3;
head->link = NULL;
```

- Since this node is the last node, the link is set to NULL.
Function head_insert

It would be better to create a function to insert nodes at the head of a list, such as:

```c
void head_insert(NodePtr& head, int the_number);
```

- The first parameter is a NodePtr parameter that points to the first node in the linked list
- The second parameter is the number to store in the list

head_insert will create a new node for the number
- The number will be copied to the new node
- The new node will be inserted in the list as the new head node
Pseudocode for head_insert

- Create a new dynamic variable pointed to by temp_ptr
- Place the data in the new node called *temp_ptr
- Make temp_ptr's link variable point to the head node
- Make the head pointer point to temp_ptr
Adding a Node to a Linked List

1. Set up new node

2. temp_ptr->link = head;

3. head = temp_ptr;

4. After function call
Translating head_insert to C++

The pseudocode for head_insert can be written in C++ using these lines in place of the lines of pseudocode:

- NodePtr temp_ptr;  //create the temporary pointer
  temp_ptr = new Node;  // create the new node
- temp_ptr->data = the_number;  //copy the number
- temp_ptr->link = head;  //new node points to first node
- head = temp_ptr;  // head points to new
                  // first node
Function to Add a Node at the Head of a Linked List

Function Declaration

```c
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;

void head_insert(NodePtr& head, int the_number);
//Precondition: The pointer variable head points to
//the head of a linked list.
//Postcondition: A new node containing the_number
//has been added at the head of the linked list.
```

Function Definition

```c
void head_insert(NodePtr& head, int the_number)
{
    NodePtr temp_ptr;
    temp_ptr = new Node;

    temp_ptr->data = the_number;

    temp_ptr->link = head;
    head = temp_ptr;
}
```
An Empty List

- A list with nothing in it is called an empty list
- An empty linked list has no head node
- The head pointer of an empty list is NULL

```c
head = NULL;
```

- Any functions written to manipulate a linked list should check to see if it works on the empty list
You might be tempted to write head_insert using the head pointer to construct the new node:

```cpp
head = new Node;
head->data = the_number;
```

Now to attach the new node to the list
- The node that head used to point to is now lost!
Lost Nodes

head

12
?

15

3
NULL

Lost nodes
Memory Leaks

- Nodes that are lost by assigning their pointers a new address are not accessible any longer.
- The program has no way to refer to the nodes and cannot delete them to return their memory to the freestore.
- Programs that lose nodes have a memory leak.
  - Significant memory leaks can cause system crashes.
Searching a Linked List

To design a function that will locate a particular node in a linked list:

- We want the function to return a pointer to the node so we can use the data if we find it, else return NULL
- The linked list is one argument to the function
- The data we wish to find is the other argument
- This declaration will work:

```c
NodePtr search(NodePtr head, int target);
```
Function search

- Refining our function
  - We will use a local pointer variable, named here, to move through the list checking for the target
    - The only way to move around a linked list is to follow pointers
  - We will start with here pointing to the first node and move the pointer from node to node following the pointer out of each node
Searching a Linked List

1. target is 6

1. head -> 2
   here -> 1
   1 -> 6
   6 -> 3 NULL

2. head -> 2
   here -> 1
   1 -> 6
   6 -> 3 NULL

3. head -> 2
   here -> 1
   1 -> 6
   6 -> 3 NULL

4. head -> 2
   here -> 1
   1 -> 6
   6 -> 3 NULL

Not found

Found
Pseudocode for search

- Make pointer variable here point to the head node
- while (here does not point to a node containing target AND here does not point to the last node)
  
  ```
  make here point to the next node
  ```

- If (here points to a node containing the target)
  
  ```
  return here;
  ```

  else
  
  ```
  return NULL;
  ```
Moving Through the List

The pseudocode for search requires that pointer here step through the list

- How does here follow the pointers from node to node?
- When here points to a node, here->link is the address of the next node
- To make here point to the next node, make the assignment:
  
  here = here->link;
A Refinement of search

The search function can be refined in this way:
here = head;
while(here->data != target && here->link != NULL)
{
    here = here->next;
}
if (here->data == target)
    return here;
else
    return NULL;
Our search algorithm has a problem

- If the list is empty, here equals NULL before the while loop so…
  - here->data is undefined
  - here->link is undefined
- The empty list requires a special case in our search function
- A refined search function that handles an empty list is shown in following
Function to Locate a Node in a Linked List

Function Declaration

```c
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;

NodePtr search(NodePtr head, int target);
//Precondition: The pointer head points to the head of
//a linked list. The pointer variable in the last node
//is NULL. If the list is empty, then head is NULL.
//Returns a pointer that points to the first node that
//contains the target. If no node contains the target,
//the function returns NULL.
```

Function Definition

```c
//Uses cstddef:
NodePtr search(NodePtr head, int target)
{
    NodePtr here = head;
    if (here == NULL)
    {
        return NULL;  // Empty list case
    }
    else
    {
        while (here->data != target &&
               here->link != NULL)
        {
            here = here->link;
        }
        if (here->data == target)
            return here;
        else
            return NULL;
    }
}
```
Pointers as Iterators

An iterator is a construct that allows you to cycle through the data items in a data structure to perform an action on each item

- An iterator can be an object of an iterator class, an array index, or simply a pointer

A general outline using a pointer as an iterator:

```c
Node_Type *iter;
for (iter = Head; iter != NULL; iter = iter->Link)
    //perform the action on the node iter points to
```

- Head is a pointer to the head node of the list
Using the previous outline of an iterator we can display the contents of a linked list in this way:

```cpp
NodePtr iter;
for (iter = head; iter != NULL; iter = iter->Link)
    cout << (iter->data);
```
To insert a node after a specified node in the linked list:

- Use another function to obtain a pointer to the node after which the new node will be inserted
- Call the pointer after_me
- Use function insert, declared here to insert the node:

  ```c
  void insert(NodePtr after_me, int the_number);
  ```
Inserting in the Middle of a Linked List
Inserting the New Node

- Function insert creates the new node just as head_insert did

- We do not want our new node at the head of the list however, so…
  - We use the pointer after_me to insert the new node
Inserting the New Node

This code will accomplish the insertion of the new node, pointed to by temp_ptr, after the node pointed to by after_me:

```
temp_ptr->link = after_me->link;
after_me->link = temp_ptr;
```
The order of pointer assignments is critical
- If we changed after_me->link to point to temp_ptr first, we would lose the rest of the list!

The complete insert function is shown in following
Function to Add a Node in the Middle of a Linked List

Function Declaration

```c
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;

void insert(NodePtr after_me, int the_number);
// Precondition: after_me points to a node in a linked list.
// Postcondition: A new node containing the_number has been added after the node pointed to by after_me.
```

Function Definition

```c
void insert(NodePtr after_me, int the_number)
{
    NodePtr temp_ptr;
    temp_ptr = new Node;

    temp_ptr->data = the_number;

    temp_ptr->link = after_me->link;
    after_me->link = temp_ptr;
}
```
Function insert Again

- Notice that inserting into a linked list requires that you only change two pointers
  - This is true regardless of the length of the list
  - Using an array for the list would involve copying as many as all of the array elements to new locations to make room for the new item

- Inserting into a linked list is often more efficient than inserting into an array
Removing a Node

To remove a node from a linked list

- Position a pointer, before, to point at the node prior to the node to remove
- Position a pointer, discard, to point at the node to remove
- Perform: before->link = discard->link;
  - The node is removed from the list, but is still in memory
- Return *discard to the freestore: delete discard;
Removing a Node

1. Position the pointer `discard` so that it points to the node to be deleted, and position the pointer before so that it points to the node before the one to be deleted.

2. `before->link = discard->link;`

3. `delete discard;`
If head1 and head2 are pointer variables and head1 points to the head node of a list:

```c
head2 = head1;
```

causes head2 and head1 to point to the same list.

There is only one list!

If you want head2 to point to a separate copy, you must copy the list node by node or overload the assignment operator appropriately.
DISPLAY 13.11  A Doubly Linked List

```
front

1

2

3

back
```
**DISPLAY 13.12  A Binary Tree**
13.2

Stacks and Queues
A stack is a data structure that retrieves data in the reverse order the data was stored.

- If 'A', 'B', and then 'C' are placed in a stack, they will be removed in the order 'C', 'B', and then 'A'.

A stack is a last-in/first-out data structure like the stack of plates in a cafeteria; adding a plate pushes down the stack and the top plate is the first one removed.
A Stack

A

B

C

B

A

C

B

A

C

B

A

52
We will create a stack class to store characters

- Adding an item to a stack is pushing onto the stack
- Member function push will perform this task

- Removing an item from the stack is popping the item off the stack
- Member function pop will perform this task
// This is the header file stack.h. This is the interface for the class Stack, 
// which is a class for a stack of symbols.

#ifndef STACK_H
#define STACK_H

namespace stacksavitch
{
    struct StackFrame
    {
        char data;
        StackFrame *link;
    };

typedef StackFrame* StackFramePtr;

class Stack
{
    public:
        Stack();  // Initializes the object to an empty stack.
        Stack(const Stack& a_stack);  // Copy constructor.
        ~Stack();  // Destroys the stack and returns all the memory to the freestore.
        void push(char the_symbol);  // Postcondition: the_symbol has been added to the stack.
        char pop();  // Precondition: The stack is not empty.
                        // Returns the top symbol on the stack and removes that 
                        // top symbol from the stack.
        bool empty() const;  // Returns true if the stack is empty. Returns false otherwise.
    private:
        StackFramePtr top;
    };///stacksavitch

#ifndef //STACK_H
Function push

The push function adds an item to the stack
- It uses a parameter of the type stored in the stack
  
  ```
  void push(char the_symbol);
  ```
- Pushing an item onto the stack is precisely the same task
  accomplished by function head_insert of the linked list
- For a stack, a pointer named `top` is used instead of a
  pointer named head
Function pop

The pop function returns the item that was at the top of the stack

char pop();

- Before popping an item from a stack, pop checks that the stack is not empty
- pop stores the top item in a local variable result, and the item is "popped" by: \( \text{top} = \text{top} \rightarrow \text{link} \);
- A temporary pointer must point to the old top item so it can be "deleted" to prevent a memory leak
- pop then returns variable result
Empty Stack

- An empty stack is identified by setting the top pointer to NULL

  top = NULL;
The Copy Constructor

Because the stack class uses a pointer and creates new nodes using new, a copy constructor is needed

- The copy constructor (a self-test exercise) must make a copy of each item in the stack and store the copies in a new stack

- Items in the new stack must be in the same position in the stack as in the original
The stack destructor

Because function pop calls delete each time an item is popped off the stack, ~stack only needs to call pop until the stack is empty

```c
char next;
while( ! empty( ) )
{
    next = pop( );
}
```
Implementation of the Stack Class (part 1 of 2)

```
//This is the implementation file stack.cpp.
//This is the implementation of the class Stack.
//The interface for the class Stack is in the header file stack.h.
#include <iostream>
#include <cstddef>
#include "stack.h"
using namespace std;

namespace stacksavitch
{
    //Uses cstddef:
    Stack::Stack() : top(NULL)
    {
        //Body intentionally empty.
    }

    Stack::Stack(const Stack& a_stack)
        <The definition of the copy constructor is Self-Test Exercise 11.>
```
Implementation of the Stack Class (part 2 of 2)

```cpp
Stack::~Stack()
{
    char next;
    while (! empty())
        next = pop(); // pop calls delete.
}

// Uses cstddef:
bool Stack::empty() const
{
    return (top == NULL);
}

void Stack::push(char the_symbol)
<The rest of the definition is Self-Test Exercise 10.>

// Uses iostream:
char Stack::pop()
{
    if (empty())
    {
        cout << "Error: popping an empty stack.\n";
        exit(1);
    }

    char result = top->data;

    StackFramePtr temp_ptr;
    temp_ptr = top;
    top = top->link;

    delete temp_ptr;

    return result;
} // stacksavitch
```
Program to demonstrate use of the Stack class.
#include <iostream>
#include "stack.h"
using namespace std;
using namespace stacksavitch;

int main()
{
    Stack s;
    char next, ans;

    do
    {
        cout << "Enter a word: ";
        cin.get(next);
        while (next != '\n')
        {
            s.push(next);
            cin.get(next);
        }
        cout << "Written backward that is: ";
        while ( ! s.empty() )
            cout << s.pop();
        cout << endl;

        cout << "Again?(y/n): ";
        cin >> ans;
        cin.ignore(10000, '\n');
    } while ( ans != 'n' && ans != 'N');

    return 0;
}

The ignore member of cin is discussed in Chapter II. It discards input remaining on the current input line up to 10,000 characters or until a return is entered. It also discards the return ("\n") at the end of the line.
Program Using the Stack Class (part 2 of 2)

Sample Dialogue

Enter a word: straw
Written backward that is: warts
Again?(y/n): y
Enter a word: C++
Written backward that is: ++C
Again?(y/n): n
DISPLAY 13.21  Interface File for a Queue Class (part 1 of 2)

1 //This is the header file queue.h. This is the interface for the class Queue,
2 //which is a class for a queue of symbols.
3 #ifndef QUEUE_H
4 #define QUEUE_H
5 namespace queuesavitch
6 {
7     struct QueueNode
8     {
9         char data;
10         QueueNode *link;
11     };
12     typedef QueueNode* QueueNodePtr;
13
14     class Queue
15     {
16         public:
17         Queue();
18         //Initializes the object to an empty queue.
19         Queue(const Queue& aQueue);
20         ~Queue();
DISPLAY 13.21  Interface File for a Queue Class (part 2 of 2)

21    void add(char item);
22    // Postcondition: item has been added to the back of the queue.
23    char remove();
24    // Precondition: The queue is not empty.
25    // Returns the item at the front of the queue and
26    // removes that item from the queue.
27    bool empty() const;
28    // Returns true if the queue is empty. Returns false otherwise.
29    private:
30    QueueNodePtr front; // Points to the head of a linked list.
31    // Items are removed at the head
32    QueueNodePtr back; // Points to the node at the other end of the
33    // linked list. Items are added at this end.
34    
35    } // queuesavitch
36    #endif // QUEUE_H
DISPLAY 13.22  Program Using the Queue Class (part 1 of 2)

1    //Program to demonstrate use of the Queue class.
2    #include <iostream>
3    #include "queue.h"
4    using namespace std;
5    using namespace queuesavitch;
6
7    int main()
8    {
9        Queue q;
10        char next, ans;
11
12        do
13            {
14            cout << "Enter a word: ";
15            cin.get(next);
16            while (next != '\n')
17                {
18                    q.add(next);
19                    cin.get(next);
20                }
21        }
DISPLAY 13.22  Program Using the Queue Class (part 2 of 2)

```cpp
22    cout << "You entered:: ";
23    while (! q.empty() )
24        cout << q.remove();
25        cout << endl;
26
27    cout << "Again?(y/n): ";
28    cin >> ans;
29    cin.ignore(10000, '\n');
30    }while (ans !='n' && ans != 'N');
31
32    return 0;
33 }
```

The `ignore` member of `cin` is discussed in Chapter 8. It discards input remaining on the current input line up to 10,000 characters or until a return is entered. It also discards the return (`'\n'`) at the end of the line.

---

**Sample Dialogue**

Enter a word: **straw**
You entered: straw
Again?(y/n): y
Enter a word: **C++**
You entered: C++
Again?(y/n): n