Linked Data Structures

A linked list is a sequence data structure that consists of Nodes:
- Node is a data structure that carries some information (likely templated).
- A Node also contains information about a successor Node, and potentially a predecessor Node.
  - could actually contain information about lots of other Nodes.

An example

- each node in the list was (more than likely) dynamically allocated.
- the memory each node uses is not contiguous.
  - there is no one solid chunk of memory but lots of little chunks of memory with pointers.
two kinds of dynamic memory

Node \( \text{Data} \rightarrow \text{ptr} \rightarrow \text{Node} \rightarrow \text{Data} \rightarrow \text{ptr} \rightarrow \text{Node} \rightarrow \text{Data} \rightarrow \text{ptr} \rightarrow \text{head} \)

vs

[Diagram of an array]

array

- O(1) index lookup
  - why is that?
- O(n) insertion
  - why is that?
- O(n) to grow the array size (to dynamically allocate a larger array)
  - again, why is that?

list

- O(n) indexing
  - why?
- O(1) insertion, deletion and Node movement
- O(1) growing size of the list
  - can grow or shrink on demand easily

Disadvantages

- You typically do not do array indexing of a list as this implies a O(n) search
  - starting from head, traverse the list to find the nth element.
- Has no capacity (no hunk of memory) but does grow and shrink
**STL list**

Example 18.1

Not surprisingly, the STL has a list container.

`list` does not support `operator[]` nor the `.at` member function. It does have a set of special member functions that other containers do not.

**STL is about efficiency**

- Could do a `operator[]` on a list. STL did not because it is not efficient!
- STL provides efficiency guarantees on its operators and does not implement operators that are inherently efficient
  - use the right container for the job!

**interesting ops (1)**

- `lst.push_front(val)`: append to the front of the list
- `lst.pop_front()`: remove from the front
- `lst.sort()`: sort the list **in place**. Not a generic algorithm, a **member function**
- `lst.reverse()`: reverse the list **in place**. Again, a **member function** not a generic algorithm
interesting ops(2)

- `lst.remove(val)` : member function that removes all examples of a value from the list.
- `lst.merge(lst2)` : takes two sorted lists and merges them into one list. `lst` grows, `lst2` is depleted (elements are moved, not copied)
- `lst.unique()` takes a sorted list and removes consecutive, equal elements

interesting ops(3)

- `lst.splice(itr, lst2)` : moves the contents of `lst2` into `lst` starting at the position of `itr` (of `lst`).
- `lst.insert(itr, val)` : inserts the value into the list at the location before `itr` (of `lst`).

only bi-directional iterators

List does not support `operator[]` so it makes sense that it also doesn't support iterators at a particular location.

Bi-directional moves you can move forward(`itr++`) or backward(`itr--`) but cannot assign (`itr = itr + 3`)

why more efficient?

These operations are more efficient because data does not have to be copied!

Instead, pointers to the data can be moved around (very cheap) to achieve the result (sorted, reversed, etc.)
Again, if you want to do linked list stuff, you should use the STL container.

We will cover how to build our own, but the STL is:
• more efficient
• proven
• does memory management.

The objects being placed in the list need certain capabilities for all the list member functions to work
• copy constructor. For example, an object inserted into a list is copied.
• operator<. For sort, the object must have a compare operator
• operator==. For unique or remove, must be able to find equal objects

Make our own

Ex 18.2
This data structure is called a **singly linked** list:
- singly linked because each Node only knows who its successor is
- has a head pointer that points to the first element of the list

A singly linked list Node has two private data members:
- a `T` `data_` member, the "payload" the Node carries
- a `Node<T> *next_` pointer. It can point to another `Node<T>`. It is used to point to the successor of this Node

The process is to link Nodes together by having each `Node.next_` point to the next/successor element.

By starting at the beginning, you can traverse the entire list by following the successive `next_` pointers.
friend class

Note that we also declare the (as yet not seen) SingleLink class as a friend of the Node class.
• we need access to those next_ptrs.

Alternatives also exist:
• Node could be a struct that is all public
  • it is just a “payload”
• Node is a private class of the encompassing list
  • can’t even see it at the programmer level, all hidden

"dummy" declaration

To declare the SingleLink as a friend of Node, we have to put a "dummy" class called SingleLink in the file.

It gets over-ridden later by the full def'n but acts as a place holder here.

SingleLink

template<typename T>
class SingleLink{
private:
  Node<T> *head_ = nullptr;
  Node<T> *tail_ = nullptr;
...
To make some operations easier, the SingleLink class maintains two private pointers:

- head_, a Node<T> ptr, the first Node
- tail_, a Node<T> ptr, the last Node

Traditionally, SingleLink has only the head_ ptr, but tail_ allows append_back with the same efficiency as append_front

public:
SingleLink()=default;
SingleLink(Node<T> n) : head_(&n), tail_(&n) {};
SingleLink(T d);
SingleLink(const SingleLink&);
SingleLink& operator=(SingleLink);
~SingleLink();
void append_front(Node<T> &n);
void append_front(T dat);
void append_back(Node<T> &n);
void append_back(T dat);
Node<T>* find(T dat);
void insert_after(Node<T> &n, Node<T> *ptr);
void insert_after(T dat, Node<T> *ptr);
friend ostream& operator<<(ostream& out,
SingleLink<T>& sl)
{
    sl.print_list(out);
    return out;
};
Constructors

SingleLink() =default;
SingleLink(Node<T> n) : head_(&n), tail_(&n) {};

Default respects the default values, both pointers to nullptr
1 param makes a list of 1 Node, both head_ and tail_ point to that Node

constructor for data

template<typename T>
SingleLink<T>::SingleLink(T d) {
    Node<T>* ptr = new Node<T>(d);
    head_ = ptr;
    tail_ = ptr;
};

append_front

template<typename T>
void SingleLink<T>::append_front(Node<T>& n) {
    if (head_ != nullptr) {
        n.next_ = head_;
        head_ = &n;
    } else {
        head_ = &n;
        tail_ = &n;
    }
}
n.next_ = head_

head_ = &n

Have to get the order right
• assign n.next_ to what head_ points to
• then change head_ to &n
• reverse it, doesn't work
  if head_ == nullptr, the list is empty
• assign both head_ and tail_ to &n
This is a constant time operation as it only involved some pointer manipulation
  • doesn't depend on the size of the list

template<typename T>
void SingleLink<T>::append_front(T dat)
{
    Node<T>* n = new Node<T>(dat);
    append_front(*n);
}

append_back
Normaly, this operation is an O(n) operation if you only have a head_ pointer
  • you have to traverse from the beginning to the end
  • then the constant overhead of the pointer manipulation
tail_ pointer makes this O(1)
What does `tail->next_` mean?

- `tail_` points to the last Node.
- `*tail_` is that node
- `(*tail).next_` is what tail has as its `next_` value (should be `nullptr`, but it can be an lvalue and set!)
- `tail_->next_` is shortcut for former

### `append_back`

```cpp
void append_back(Node n)
{
    tail_ = &n;
    tail_->next_ = nullptr;
}
```

### `tail_ = &n`

```cpp
Node tail_ = &n;
```
template<typename T>
void SingleLink<T>::append_back(T dat)
{
    Node<T>* n = new Node<T>(dat);
    append_back(*n);
}

insert(Node<T> &n, Node<T> *ptr)
insert n into the list after ptr.
• can't insert before head_, hence
  append_front
Time to find ptr is not included as part of insert.
• find would be O(n) time
Therefore insert is also O(1), just
pointer manipulation

template<typename T>
void SingleLink<T>::insert(Node<T> &n, Node<T> *ptr)
{
    if (ptr != nullptr)
    {
        n.next_ = ptr->next_;  // Increment the pointer
        ptr->next_ = &n;
        if (ptr == tail_)
            tail_ = &n;
    }
}
If `ptr` points to the same Node as `tail_`, we must also update `tail_`.

```cpp
template<typename T>
Node<T> * SingleLink<T>::find(T dat){
    Node<T> *result = nullptr;
    for(Node<T> *n = head_;
        n != nullptr;
        n = n->next_){
        if (n->data_ == dat){
            result = n;
            break;
        }
    // of if
    }
    // of for
return result;
}
```
Singly linked list can only be traversed in one direction
• you can only iterate from the beginning through the elements
• can be a restriction
• STL has a `forward_list` that is forward iteration only

```cpp
template<typename T>
SingleLink<T>::SingleLink(const SingleLink& sl) {
    if (sl.head_ == nullptr) {
        head_ = nullptr;
        tail_ = nullptr;
    } ...
```

```cpp
template<typename T>
void SingleLink<T>::insert_after(T dat, Node<T>* ptr) {
    Node<T>* n = new Node<T>(dat);
    insert_after(*n, ptr);
}
```
```cpp
else{
    head_ = new Node<T>(sl.head_->data_);
    tail_ = head_;  
    Node<T>* sl_ptr = sl.head_->next_;  
    Node<T>* new_node;

    while (sl_ptr != nullptr){
        new_node = new Node<T>(sl_ptr->data_);
        tail_->next_ = new_node;
        sl_ptr = sl_ptr->next_;  
        tail_ = new_node;
    }  // of while
}  // of else
}  // of constructor
```
Copy and Swap

```
template<typename T>
SingleLink<T>&
SingleLink<T>::operator=(SingleLink<sl>){
    swap(head_, sl.head_);
    swap(tail_, sl.tail_);
}
```

Call to copy ctor

sl destroyed at end of call (cleans up old memory)

Destructor

```
template<typename T>
SingleNode<T>::~SingleLink()
{
    Node<T>* to_del = head;
    while (to_del != nullptr){
        head_ = head_->next;
        delete to_del;
        to_del = head;
    }
    head_ = nullptr;
    tail_ = nullptr;
}
```

```
Node<T>* to_del = head;
while (to_del != nullptr){
    // one pass
    head_ = head_->next;
    delete to_del;
    to_del = head;
}
```
Doubly linked list

Example 18.4

A doubly linked list Node has two pointers:
- a successor pointer
- a predecessor pointer

This is what the STL list has, and allows for movement forward and backwards through the list.
- hence bi-directional iterators

```cpp
template<typename T>
class Node{
private:
    Node *next_;  // successor pointer
    Node *prev_;  // predecessor pointer
    T data_;      // data
    string tail_4(Node<T> *ptr);

public:
    Node() : next_(nullptr), prev_(nullptr) {};
    Node(T d) : data_(d), next_(nullptr), prev_(nullptr) {};
    ostream& print_node(ostream&);  // print node
friend class DoubleLink<T>;
};
```
private function to print the last 4 values (in hex) of an address

- makes it easier to read
- allows you to print more out
- just a detail

```cpp
// insert new node n after *ptr
template<typename T>
void DoubleLink<T>::insert(Node<T> &n, Node<T> *ptr)
{
    if (ptr != nullptr)
    {
        n.next_ = ptr->next_;  
        n.prev_ = ptr; 
        (ptr->next_)->prev_ = &n; 
        ptr->next_ = &n; 
        if (ptr == tail_)
            tail_ = &n; 
    }
}
```
We need to find the prev pointer of the after node.

- we need to have after.prev point at n

In stages:
- ptr, points to the before node
- ptr->next_, the after node
- (ptr->next_)->prev_, the after node's prev pointer