Dynamic Data Structures

- We have seen that the STL containers `vector`, `deque`, `list`, `set` and `map` can grow and shrink dynamically.
- We now examine how some of these containers can be implemented in C++.
- To simplify the syntax, we assume that the data type to be stored in the containers are `int`. But this can be generalized using templates.

Linked Lists

- In a linked list, each element has two components:
  - the data
  - a pointer to the next element
- If the pointer to the next element is null, then there is no next element (i.e. the end of the list).
- To access the entire list, we need a pointer to the first element (the head of the list).
- An element is also often called a **node**.

The List Element

```
class Element {
public:
  int data;             // the data to store
  Element *next;        // pointer to next element
  Element(Element *n, int d) : next{n}, data{d} { }
};

Element *first;         // first element in the list
```

Operations on Linked Lists

- To initialize an empty list, simply set `first = nullptr`.
- To add an element to the beginning of the list:

```
first = new Element(first, data);
```

- To traverse a list from beginning to end:

```
for (Element *p = first; p != nullptr; p = p->next)
  cout << p->data << ' ';
cout << endl;
```

- Be sure not to dereference a null pointer:

```
for (Element *p = first; p && p->data != 10; p = p->next)
  ;
// p points to the first occurrence of 10 in the list.
// What happens if we do not check if p is not null?
```

Inserting Elements into Linked Lists

- We have seen how to insert elements to the beginning of the list.
- To insert at the end, we have to find the last element in the list (if there is one), and insert a new element:

```
Element *p;
for (p = first; p->next; p = p->next)
  ;
p->next = new Element(nullptr, data);
```

- What happens if the list is empty?
- To insert a new element **after** the element pointed to by `p`:

```
p->next = new Element(p->next, data);
```

- How do you insert an element **before** `p`?

Removing Elements from Linked Lists

- To remove the first element (assume list is nonempty):

```
Element *temp = first;  // need to save it first
first = first->next;
delete temp;
```

- To remove the element **after** the one pointed to by `p` (if it exists):

```
Element *temp = p->next;
p->next = p->next->next;
delete temp;
```

- How do remove a specific element or the last element?

Doubly Linked Lists

- Each element contains pointers to the next element and also to the previous element (nullptr for first element in the list).
- To avoid handling empty lists as a special case, it is common to insert a **dummy node**:

```
// data is not used
first = new Element(nullptr, nullptr, data);
first->next = first->prev = first;
```

- To see if the list is empty, check that `first->next == first`.
- The list is circular.

Inserting Elements

- To insert an element **before** the one pointed to by `p` (assume `p != first`):

```
Element *e = new Element(p, p->prev, data);
p->prev->next = e;
p->prev = e;
```

- To insert an element **after** the one pointed to by `p`:

```
Element *e = new Element(p->next, p, data);
p->next->prev = e;
p->next = e;
```

Removing Elements

- To remove the element pointed to by `p` (assume `p != first`):

```
p->prev->next = p->next;
p->next->prev = p->prev;
delete p;
```

Linked Lists as Objects

- Private member: `first` to point to dummy node.
- Constructor allocates dummy node.
- Destructor removes every node.
- Copy constructor and assignment operator needs deep copying.

Linked List: Efficiency

- Each of the following operations are "fast": insert, remove, move one element forward/backward.
- Here, "fast" means that regardless of how many elements there are in the list, each operation takes about the same amount of time.
- For containers such as vectors, inserting an element in the middle may require all elements to be moved: it gets slower as the number of elements increases.

Stacks and Queues

- Stacks and queues can be implemented using singly linked lists.
- Stacks:

  Push: add element to the beginning of list

  Pop: remove first element of the list

  Top of stack = beginning of list.
- Queues (store pointers to head and tail of list):

  Initialization: `head = tail = nullptr;`

  Push: add element to end of list (after `tail` if not `nullptr`)

  Pop: remove first element

  `head/tail` needs to be adjusted after each operation.

Iterators for Doubly Linked Lists

- An iterator class can be declared within a list class (nested class).
- Only private member is a pointer to the element.
- `++` and `--`: overloaded to move to next/previous element. Note that there are both pre-increment and post-increment.
- `*`: overloaded to return a reference to the data.
- `begin(), end()`: `first->next` and `first`.
- You can declare reverse iterators too.