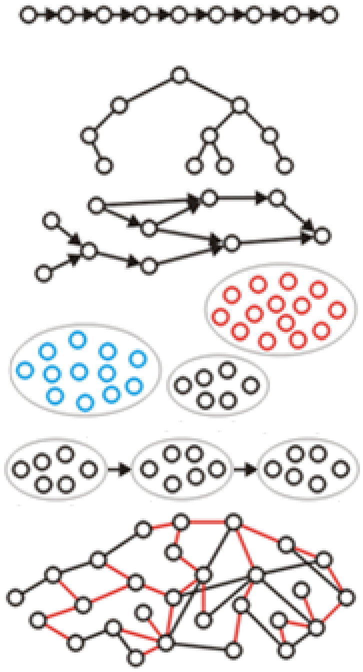
Sequential Containers – Cont'd



Carlos Moreno
cmoreno@uwaterloo.ca
EIT-4103

https://ece.uwaterloo.ca/~cmoreno/ece250

Sequential Containers

Standard reminder to set phones to silent/vibrate mode, please!

Sequential Containers

- Today's class:
  - We'll complete our discussion on sequential containers
  - We'll briefly talk about linked lists
  - Look into the notion of iterators (an example of a Design Pattern)

Sequential Containers

- Linked lists:
  - As mentioned, one of the simplest examples of Node-based containers.
  - You're already familiar with the two basic types: singly-linked lists and doubly-linked lists.

Sequential Containers

- Linked lists:
  - How do they compare to arrays?
    - Storage: less efficient; there is per-element overhead (the link to the next, or links to next and previous, elements)
    - Access: Depending on the type of access, each one has its own advantages:
      - Insertion of elements is more efficient for linked lists ($\Theta(1)$ vs. $\Theta(n)$ for arrays — average $\Theta(1)$ if using exponential growth; but still, some insertions are expensive, and this may be a problem for some types of applications; plus, that's only appends!)
      - Removal of elements — $\Theta(1)$ for linked lists; for arrays, it is $\Theta(1)$ if removing the last element; removal at arbitrary positions takes $\Theta(n)$

Sequential Containers

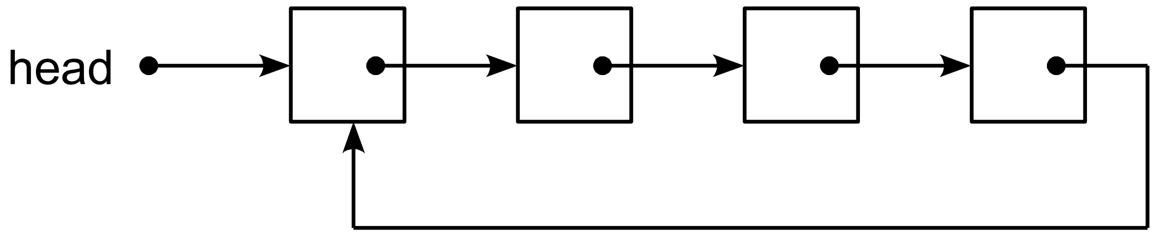
- Linked lists:
  - How do they compare to arrays?
    - Access (cont'd):
      - Random access to elements (i.e., subscripted access, as opposed to sequential access): arrays win here, of course — $\Theta(1)$ vs. $\Theta(n)$ for linked lists.

Sequential Containers

- Linked lists:
  - In terms of implementation details, there are a couple of often used features (could even think of them as “tricks”):
    - Circular list
    - With dummy or sentinel element or node.

Sequential Containers

- Circular Linked lists:
    - Instead of pointing to NULL to designate the end, the last node points back to the first one.

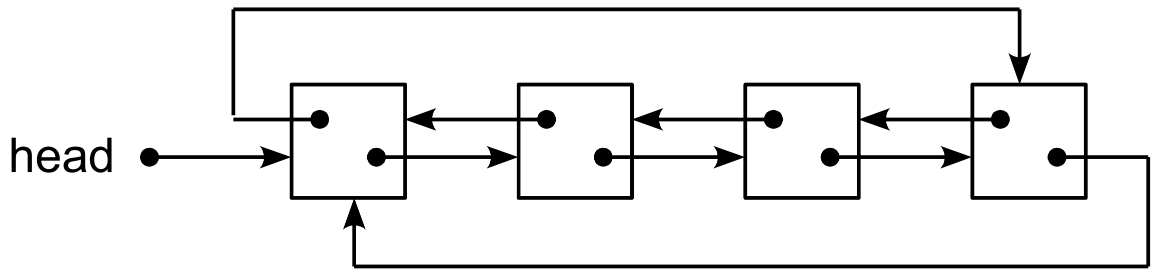


Sequential Containers

- Circular Linked lists:
  - This can simplify the coding of some operations, in that there are fewer exceptional cases to handle (such as insertion at the end), since there is always a next element.
  - Can also simplify client code when representing something that is “closed” or circular by nature; for example, using a linked list of points (vertices) to represent closed polygons (after the last vertex, the sequence “closes” going back to the first one).

Sequential Containers

- Circular Linked lists:
  - For doubly-linked lists, of course the first node's previous will point to the last one (instead of pointing to NULL)



Sequential Containers

- Dummy or Sentinel node:
  - A common trick to simplify the implementation is to add an extra (“dummy”) node at the beginning and/or another one at the end (the same one if the list is circular).
  - Thus, an empty list does have one element, avoiding the need for special code to handle insertions on empty lists.

Sequential Containers

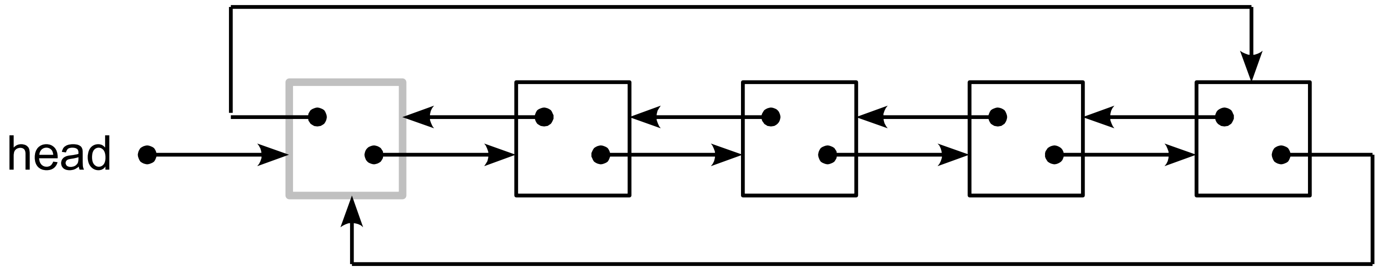
- Dummy or Sentinel node:
  - Not only a matter of simplifying the code — in terms of efficiency, we get an important advantage:

    If there is a special case that needs to be handled, we have to always check for that special case — even if it happens very rarely.

    This is pretty bad, since we have an extra cost in *every* single operation, even if only once in a while the condition applies.

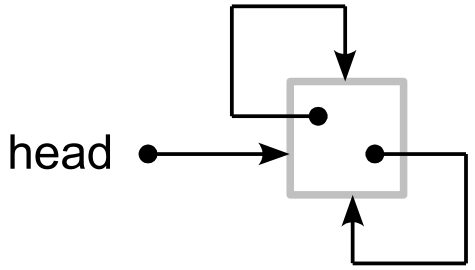
Sequential Containers

- Dummy or Sentinel node:
  - A very common approach is implementing lists as circular lists with dummy or sentinel node



Sequential Containers

- Dummy or Sentinel node:
  - A neat detail is that this is what an empty list looks like:



  - That is, inserting an element on an empty list or removing an element causing the list to become empty are operations identical to their non-empty counterpart (i.e., require no special code to handle)

Sequential Containers

- Another neat trick that can be done with a sentinel node:
  - When searching for a value, our loop has to check for the value and also whether the pointer is NULL (if the value is not found, we will reach the end of the list)

```
for (Node * p = list.head();
            p != NULL && p->retrieve() != value;
            p = p->next())
```

Sequential Containers

- Notice that this is a per-pass overhead (if we have 1000 nodes, then 1000 times we check for NULL, even though only one of them will be NULL)
- If we have a sentinel node, we can write the value we're searching, and that way we know that it will be found — no need to check for NULL at every step (at the end, check if we're pointing to the sentinel, which would mean that the value was not found in the list)

Sequential Containers

- Both queues and stacks can be conveniently implemented in terms of a linked list:
    - For queues: enqueue is implemented as push_front and dequeue is implemented as pop_back (or vice-versa)
    - For stacks: push is implemented as push_xxx and pop is implemented as pop_xxx (where xxx can be either front or back — the important detail is that it be the same for both operations)

Sequential Containers

- Next, we'll look into the notion of iterators

Sequential Containers

- Next, we'll look into the notion of iterators

  A very relevant quote, by David Wheeler (first person to ever be awarded a PhD in Computer Science, back in 1951):

  Every problem in computer science can be solved by an extra level of indirection... Except for ....

Sequential Containers

- Next, we'll look into the notion of iterators

  A very relevant quote, by David Wheeler (first person to ever be awarded a PhD in Computer Science, back in 1951):

  Every problem in computer science can be solved by an extra level of indirection... Except for the problem of too many levels of indirection

  – David Wheeler

Sequential Containers

- Question: since both arrays and linked lists are sequential containers, shouldn't we be able to sequentially access the element in the exact same way?
  - «*Why would we want to do that ?*», one could ask.
  - What happens if we choose, say, arrays since there are few insertions or removals, but then the requirements are changed (or re-evaluated) and we decide that a linked list does a better job?

Sequential Containers

- Question: since both arrays and linked lists are sequential containers, shouldn't we be able to sequentially access the element in the exact same way?
  - «*Why would we want to do that ?*», one could ask.
  - What happens if we choose, say, arrays since there are few insertions or removals, but then the requirements are changed (or re-evaluated) and we decide that a linked list does a better job?
    - We'd have to re-write all code that accesses the elements in the sequence.

Sequential Containers

- Another question (completely independent and different question, yet surprisingly pointing to a common answer!):
- What about the way that we access elements in a linked list? Isn't class List (whatever its name) supposed to encapsulate and hide implementation details from client code?
  - One of the reasons for this is that if you change your implementation details, client code does not need to be changed everywhere a list is used.

Sequential Containers

- But also, we'd like, as much as possible, to protect the internals of our list class from mistakes made by outsiders.
- For example, we have a method head() that returns a Node * pointing to the first element (or first after the sentinel, if applicable)
  - What if client code thinks they can do arithmetic with that pointer? list.head()++ or something like that.

Sequential Containers

- What if client code does something like this?

```
Node * head = list.head();
// ...
delete head;
```

- There's nothing the class can do to prevent this if it hands out a pointer to a node directly.

Sequential Containers

- What if instead of returning a Node * directly, that Node * was encapsulated in a class?
- A class that only supports retrieval of the value and advancing to the next element? (or moving backward as well, if a doubly-linked list)

Sequential Containers

- This is, roughly speaking, the main idea behind the iterator design pattern:
  - It provides that extra level of indirection (a “middle-man”) that addresses these two problems that we mentioned.
  - Client code does not know about Node; it knows about List_iterator (or just Iterator), and it knows that with an iterator, it can retrieve values, advance to the next element, and check whether we're at the end of the list (to stop the loop that iterates over the sequence of values).

Sequential Containers

- The interesting aspect is that there could be a class Iterator, with exact functionality, for arrays.
  - If we only need sequential access, and not random access (i.e., subscripted), we could use the exact same code to iterate over the elements in an array or in a linked list.

Sequential Containers

- The internals of the implementations of these two iterator classes can be as radically different as needed
- As long as they both provide the same interface, and of course, as long as they both provide the correct functionality, client code will be happy!

Sequential Containers

- An example (extremely simplified) for linked list:

```
class List_iterator
{
public:
    List_iterator (Node<Type> * node)
        : d_node(node) {}
    void advance();
    Type retrieve() const;
    // ...
private:
    Node<Type> * d_node;
};
```

Sequential Containers

- Methods implementation:

```
void List_iterator<Type>::advance()
{
    d_node = d_node->next();
}

Type List_iterator<Type>::retrieve() const
{
    return d_node->retrieve();
}
```

Sequential Containers

- Now (also extremely simplified) for array:

```
class Array_iterator
{
public:
    Array_iterator (Array<Type> & array,
                                int index)
        : d_array(array), d_index(index) {}
    void advance();
    Type retrieve() const;
private:
    Array<Type> d_array;
    int d_index;
};
```

Sequential Containers

- Methods implementation:

```
void Array_iterator<Type>::advance()
{
    ++d_index;
}

Type Array_iterator<Type>::retrieve() const
{
    return d_array[d_index];
}
```

Sequential Containers

- Despite the rather important difference in the internals for both iterators, with either one, client code does the same to iterate over the sequence (since the interface is the same):

```
for (XXX_iterator i = container.begin();
                      !i.at_end();
                      i.advance())
{
    cout << i.retrieve() << endl;
}
```

Sequential Containers

- In the example, XXX would be List or Array, depending on which one we use.
- `container` is either the linked list or the array for which we want to iterate over its elements (in sequence)
- Presumably, we add the requirement that both linked list an array must provide a method `begin()`, that returns an iterator referring to the first element.

Sequential Containers

- If you're curious and interested in knowing more, you can check my tutorial on the Standard Template Library (STL) — in the course web page, check the C++ option in the menu on the left (this is completely optional!).
- STL iterators are different with respect to the ones presented here, but the idea and their purpose is essentially identical.
- Again: completely optional (not part of the course material)

Summary

- During today's class, we discussed:
  - Linked lists and some implementation tricks:
    - Circular linked lists
    - Dummy or sentinel node
  - Using a linked list to implement queues or stacks
  - Iterator as a nice design pattern addressing at least two issues with sequential containers (one of the issues being specific to linked lists)