CPSC 131
Data Structures Concepts

Dr. Anand Panangadan
apanangadan@fullerton.edu

What is a data structure?

- A way of organizing computer memory such that data operations can be performed efficiently
- Some data operations
  - Insert data
  - Remove data
  - Get specific data item
  - Get all data items

A data structure in C++

- A template class

```
template <typename E>
class DataStructure {
public:
   DataStructure();     // create an empty data
structure
   ~DataStructure();        // destructor
   bool empty() const; // is it empty?
   E& getData();
   void insertData(const E& e);
   void removeData();
   int size() const;    // how many data
elements
private:
   // whatever needed to implement the above
};
```

Singly Linked List in C++

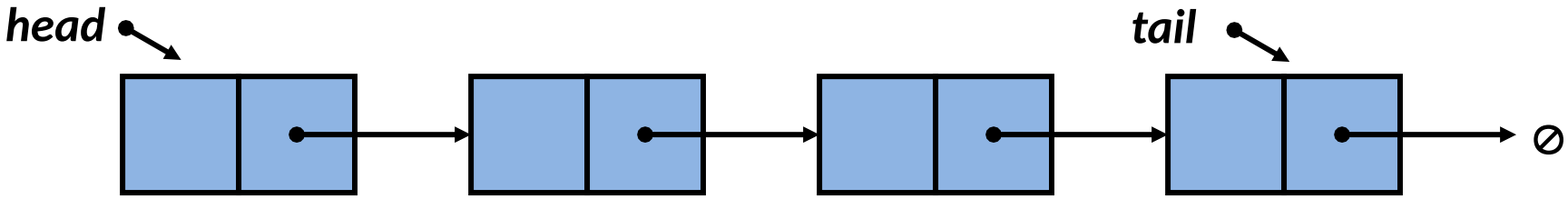
```
template <typename E>
class SinglyLinkedList {  // a singly linked list
public:
    void append(  const T& );
    void prepend( const T& );
    void insertAfter( Node<T>*,  const T& );
    void removeAfter( Node<T>* );
    void pop_front();  // remove element at front of list
    T&   front();      // return list's front element
    T& back(); // return list's back element
private:
   // whatever needed to implement the above (which is?)
};
```

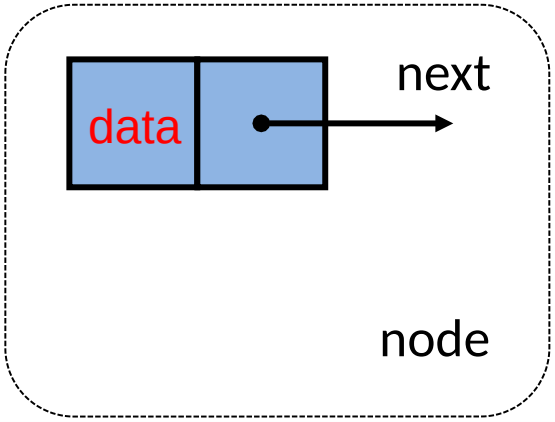
A data structure in C++

```
template <typename E>
class DLinkedList {    // a doubly linked list
public:
   DLinkedList();  // empty list constructor
   ~DLinkedList(); // destructor
    bool empty();

    void append(  const T& );
    void prepend( const T& );
    void insertAfter( Node<T>*,  const T& );
    void remove( Node<T>* );

    void pop_front();      // remove element at front of list
    void pop_back();      // remove element at back of list
    T&   front();          // return list's front element
    T& back(); // return list's back element
private:
     // local type definitions
};
```

Singly Linked List and Doubly Linked List

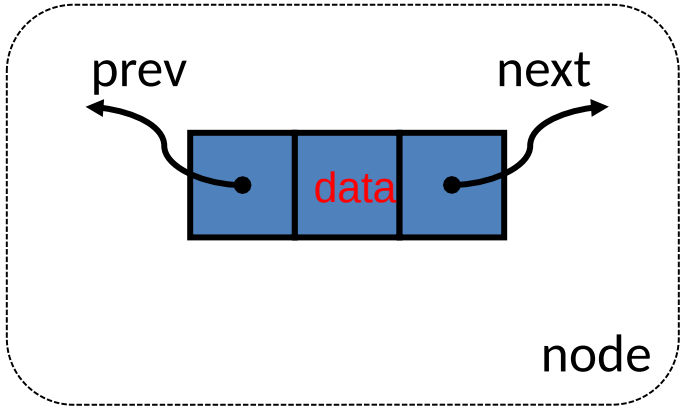
What if want to access data in reverse order?





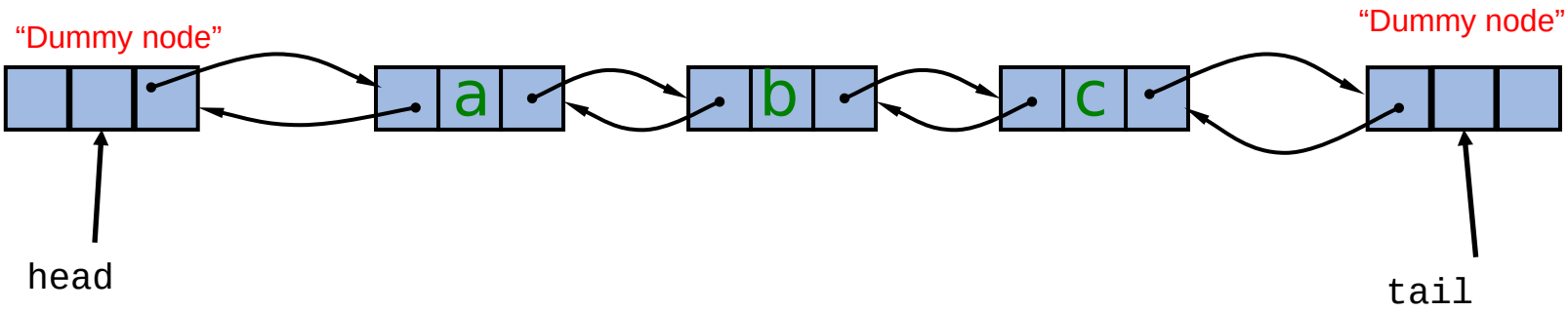
Node:
- element
- next pointer
- previous pointer

Linked List:
- length
- Head
- Tail



Doubly linked lists

- Key ideas
  - Keep a previous pointer *in addition to a next pointer* at every node
  - Keep dummy nodes at the head and tail



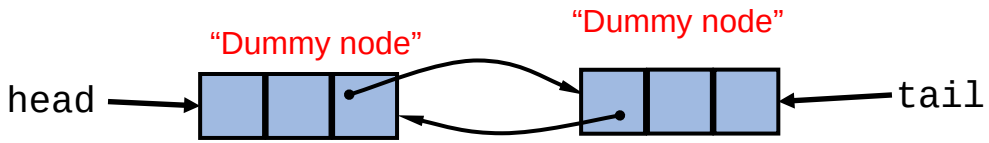
```
template<typename T>
struct Node // can also be class with all members public
{
    Node() : next(nullptr), prev(nullptr) { };
    T data; // node element value
    Node<T>* next; // next node in the list
    Node<T>* prev; // previous node in the list
};
```

Doubly Linked List constructor

- Create sentinels and interlink them

```
template <typename E>
DoublyLinkedList<E>:: DoublyLinkedList<E() { // constructor
   head = new Node<E>; // create dummy nodes
   tail = new Node<E>;
   // have them point to each other
   head->next = tail;
   tail->prev = head;
}
```

- Empty linked list: only has two dummy nodes



Generic Doubly Linked List Implementation

- https://github.com/CSUF-CPSC-131-Fall2019/Data-Structures-Code
- DoublyLinkedList.hpp
- DoublyLinkedList_main.cpp

Containers and Iterators

- **Container** is an abstract data structure that stores a collection of elements
- **Iterator** abstracts the process of looping through the collection of elements
- Let V be a container and p be an iterator for V

```
for (p = V.begin(); p != V.end(); ++p) {
        (*p).do_something()
}
```

STL Containers and Iterators

- Each STL container has an associated class Iterator
  - begin(): returns an iterator to the first element
  - end(): returns an iterator to an imaginary position just after the last element
- An iterator behaves like a pointer to an element
  - *p: returns the element referenced by this iterator; access current element
  - ++p or p++: advances to the next element
- Most STL containers provide the ability to move backwards
  - --p or p--: moves to the previous element

Iterating through Containers

- Let V be a container and p be an iterator for V

```
for (p = V.begin(); p != V.end(); ++p) {
    loop_body
}
```

STL Vector example

```
#include <vector>
using namespace std;
main() {
  vector<int> V;
  // code to insert values into V
  int sum = 0;
  for (vector<int>::iterator p=V.begin(); p != V.end(); ++p) {
     sum = sum + (*p);
}
```

STL single linked list example

```
#include <forward_list>
using namespace std;
main() {
  forward_list<int> V;
  // code to insert values into V
  int sum = 0;
  for (forward_list<int>::iterator p=V.begin(); p !=   V.end(); ++p) {
     sum = sum + (*p);
}
```

Very similar!

STL Iterators in C++

- Each STL container type C supports iterators:
  - C::iterator – read/write iterator type
  - C::const_iterator – read-only iterator type
  - C.begin(), C.end() – return start and end iterators, respectively
- Various notions of iterator:
  - ***(standard) iterator***: allows read-write access to elements
  - ***const iterator***: provides read-only access to elements
  - ***bidirectional iterator***: supports both ++p and --p

Example of Iterator *Use*

```
list<string> mylist;
list<string>::iterator myiterator;

myiterator = mylist.begin();
while (myiterator != mylist.end()) {
   cout << *myiterator << endl;
   ++myiterator;
}

for (auto myiterator = mylist.begin();
myiterator != mylist.end(); ++myiterator) {
   cout << *myiterator << endl;
}
```