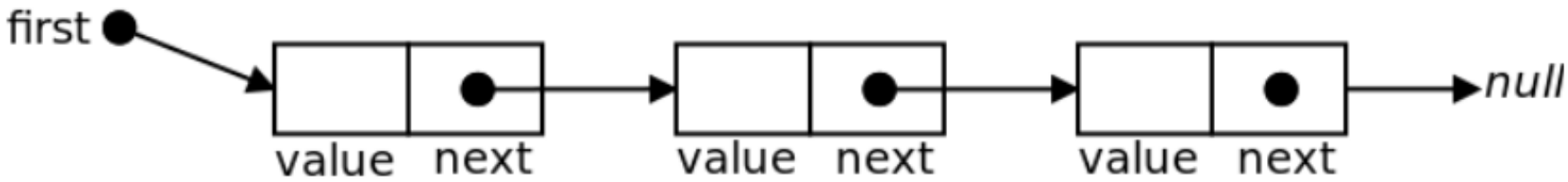
Linear Data Structures: Linked Lists

L.EIC

Algoritmos e Estruturas de Dados

2024/2025



P Ribeiro, AP Tomás

Abstract Data Types (ADT)

- Abstraction is a fundamental concept in computer science and software engineering, namely within the **object-oriented programming paradigm**.
- **Abstract Data Type**: A type of organization of data that we define through its behavior and operations.

Barbara Liskov, 1974.

" What we desire from an abstraction is a mechanism which permits the expression of relevant details and the suppression of irrelevant details. In the case of programming, the use which may be made of an abstraction is relevant; the way in which the abstraction is implemented is irrelevant."

Turing award in 2008

https://en.wikipedia.org/wiki/Barbara_Liskov

Abstract Data Types (ADT)

Data Models

- arrays
- lists
- linked lists
- trees
- stacks
- queues
- sets
- dictionaries
- hash tables
- priority queues
- binary heaps
- disjoint sets

An ADT can be implemented in many ways through different data structures

ADTs vs Data Structures

- **Abstract Data Type (ADT):**
  - *A definition for expected operations and behavior*
  - A mathematical description of a collection with a set of supported operations and how they should behave when called upon
  - Describes what a collection does, not how it does it
  - Can be expressed as an interface
  - Examples: List, Map, Set

- **Data Structure**
  - *A way of organizing and storing related data points*
  - An object that implements the functionality of a specified ADT
  - Describes exactly how the collection will perform the required operations
  - Examples: LinkedIntList, ArrayIntList

https://courses.cs.washington.edu/courses/cse373/23su/lectures/lecture01.pdf

Procedural and data abstractions

- **Procedural abstraction:**
  - Abstract from details of procedures (e.g., methods)
  - Specification is the abstraction
  - Abstraction is the specification
  - Satisfy the specification with an implementation
- **Data abstraction:**
  - Abstract from details of data representation
  - Also a specification mechanism
  - A way of thinking about programs and design
  - Standard terminology: **Abstract Data Type**, or **ADT**

ADTs

ADTs support abstraction, encapsulation, and information hiding.

- **Abstraction:** Client insulated from details, works at higher-level
- **Encapsulation:** Objects combine **data** and **operations**.
  Object internals can be kept private and secure
- **Information Hiding:** Objects hide inner details
  Internals private to ADT, not accessible by client
- **Independence / Modularity:** Separate tasks for each side (once agreed on interface)
- **Flexibility/Adaptability:** ADT implementation can be changed without affecting client

These principles help improve **robustness**, **adaptability** and increase the potential for **code reuse**.

ADTs in C++

In C++, a **class** represents an ADT. A C++ class includes both **member variables**, which define the data representation, and **member functions** that operate on the data.

- Classes consists of members (encapsulation):
  - Data members (member variables): also called data fields or attributes.
  - Member functions: also called operators, functions or methods.
- Provide information hiding by:
  - access specifiers: `private` versus `public` or `protected`
  - separating the code for the **interface** from the code for the **implementation**
    - ★ *header file* **.h** contains the member declarations
    - ★ *source file* **.cpp** contains their definitions

Remembering classes

```
#include <string>
using namespace std;

class Person {
private:
    string id;
    string name;
    int age;
public:
    Person(string id, string n="UNDEFINED", int a=0);
    string getId() const;
    string getName() const;
    int getAge() const;
};

Person::Person(string i, string n, int a) :
         id(i), name(n), age(a) {}
string Person::getId() const {return id;}
string Person::getName() const {return name;}
int Person::getAge() const {return age;}
```

Remembering classes

Header file person.h

```
#ifndef Person_H
#define Person_H

#include<string>
using namespace std;

class Person {
private:
    string id;
    string name;
    int age;
public:
    Person(string id, string n="UNDEFINED", int a=0);
    string getId() const;
    string getName() const;
    int getAge() const;
};
#endif
```

Remembering classes

Source file person.cpp

```
#include <string>

#include "person.h"
using namespace std;

Person::Person(string i, string n, int a) :
        id(i), name(n), age(a) {}

string Person::getId() const {return id;}

string Person::getName() const {return name;}

int Person::getAge() const {return age;}
```

Remembering classes

Example of usage example.cpp

```
#include <iostream>
#include <vector>

#include "person.h"

int main(){
  Person p1("12345", "John", 42);
  Person p2("42424", "Mary", 37);
  Person p3("55555", "Chariloe", 61);

  std::vector<Person> v = {p1, p2, p3};
  for(auto it = v.begin(); it != v.end(); it++)
    std::cout << (*it).getName() << std::endl;
    // std::cout << it -> getName() << std::endl;

  return 0;
}
```

Remembering classes

Compiling in Linux with g++

```
$ ls
example.cpp  person.cpp  person.h
$ g++ -c -std=c++11 person.cpp
$ ls
example.cpp  person.cpp  person.h  person.o
$ g++ -c -std=c++11 example.cpp
$ ls
example.cpp  person.cpp  person.h  person.o  example.o
$ g++ -o example example.o person.o
$ ls
example.cpp  person.cpp  person.h  person.o  example.o example
$ ./example
John
Mary
Chariloe
```

We can also use a makefile or compile altogether:

```
g++ -std=c++11 -o example example.cpp person.cpp
```

In AED, we will be using C++17:

```
g++ -std=c++17 -o example example.cpp person.cpp
```

ADT: List

- A **list** is a sequence of items of the same type.

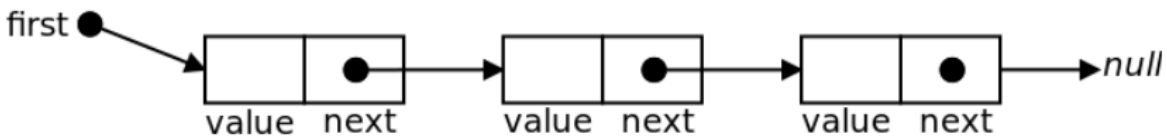
$$A_0, A_1, A_2, \ldots, A_{n-1}$$

- empty list: if it has no elements.
- Basic operations over a list:
  - create a new, empty list
  - add/remove an element from the list
  - find the position of an element in the list
  - find the *length* of the list, i.e., the number of elements
  - concatenate two lists
  - (delete/destroy the list, if it was dynamically allocated)

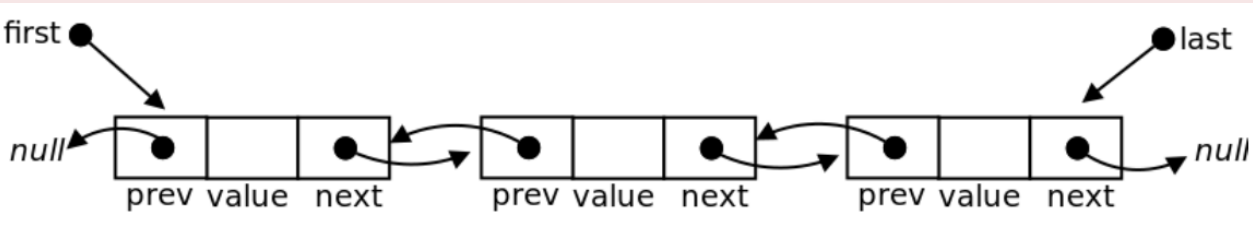
Implementing lists

Array-based lists and **linked lists** are two different underlying low-level data structures to store the element values of a list. Sequence abstractions based on **contiguous memory**, like arrays or vectors, have the drawback of inefficient insertion/deletion at the beginning or middle of the sequence (i.e., $\mathcal{O}(size)$).

A **linked list** holds a sequence of elements all of the same type, just like an **array** or a **vector** (i.e., a dynamic array). But linked lists can grow and shrink more efficiently. In a **singly-linked list**, each value is stored in one **node**. The nodes form a chain: each node contains a reference to the node containing the next element in the list.



In a **doubly-linked list**, each node contains a reference to the previous node in the list also.



Implementing lists

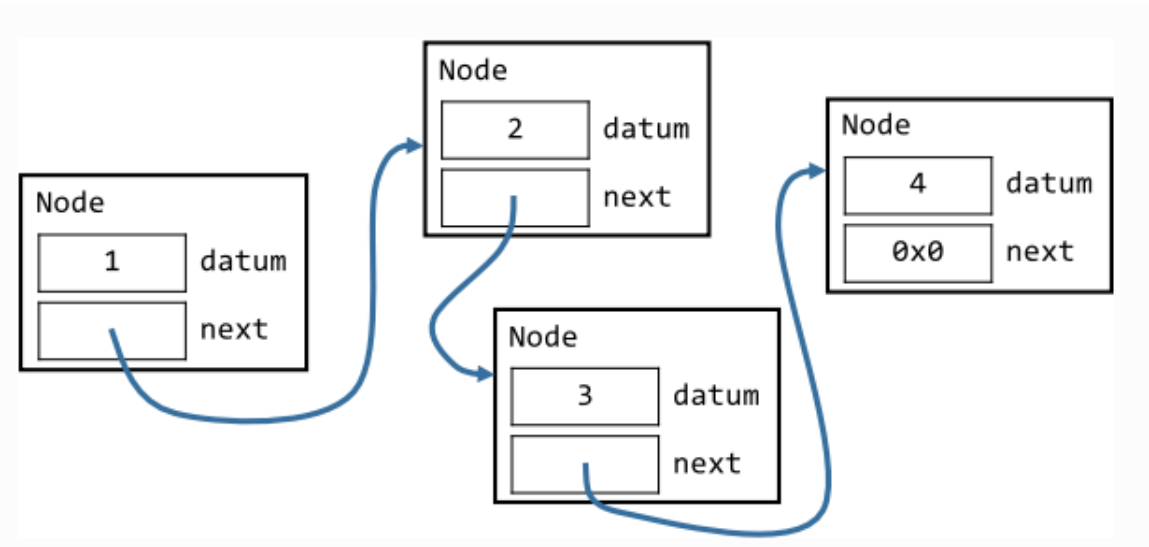
Data Structures: Array-based Lists vs Linked-lists

- An array/vector stores elements in **contiguous memory** positions.
  - Some advantages:
    - ★ Easy to use
    - ★ Quick access to a given position (random access `a[i]`)
    - ★ Arrays do not require extra storage for links (pointers)
  - Some disadvantages:
    - ★ For *arrays*, size is fixed at creation of the array (increasing implies copying elements); for *vectors*, which are dynamic allocated arrays, that operation is dynamically supported.
    - ★ Insertions and removals can be costly (many shifts of elements)
- A linked-list stores elements in **non-contiguous memory** positions.
  - Some advantages:
    - ★ Never full (if there is more memory)
    - ★ No shifts when a value is inserted/deleted
  - Disadvantages:
    - ★ Sequential access only (must be traversed to get to i-th element)

Implementation of a Linked List of Integers

Class `IntList`

Presentation based on lecture notes by Amir Kamil, Programming and Data Structures (2024) https://eecs280staff.github.io/notes/18_Linked_Lists.html#



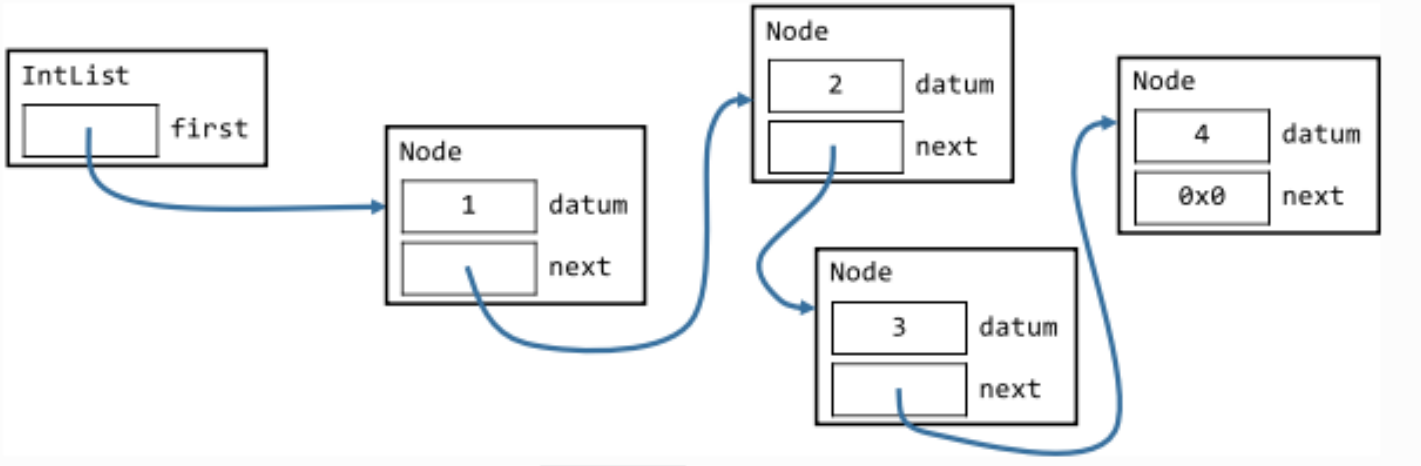
Defining a **node** for a singly-linked list:

```
struct Node {
  int datum;
  Node *next;
};
```

Implementation of a Linked List of Integers

Class `IntList`

```
class IntList {
  ...
private:
  Node *first;
};
```



An object of the class `Intlist` contains a pointer `first` to the first node in the list.

When `first` is **nullptr** (i.e., the value zero, `0x0`), the list is empty.

IntList ADT

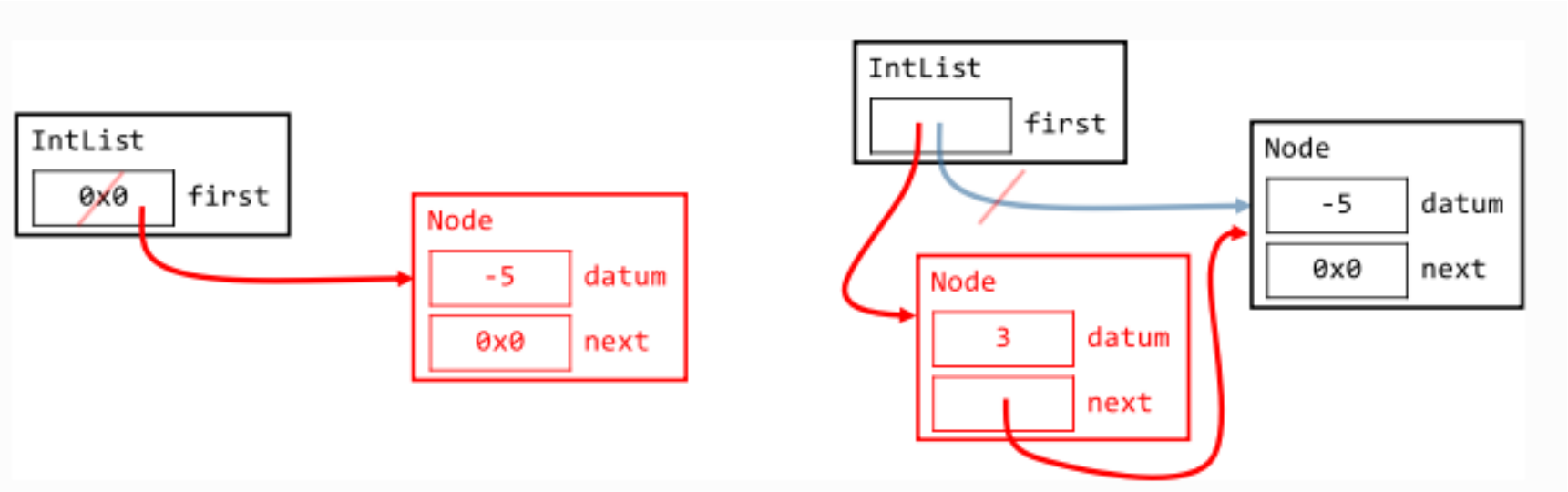
```
class IntList {
public:
  IntList();                  // Constructs an empty list.
  bool empty() const;  // Returns true if the list is empty.
  int & front();       // Returns (by reference) the first value.
  void push_front(int datum); // Inserts datum at the front.
  void pop_front();    // Removes the first element.
  void print(std::ostream &os) const;  // Prints all items.

private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;
};
```

In this implementation, `Node` appears nowhere in the interface. So, it is defined as a **private** member of `IntList`. This makes it a *nested class*, which is a class (or struct) defined as a member of another class.

IntList

`push_front` – to insert a node at front




```
void IntList::push_front(int datum) {
  Node *p = new Node;
  p->datum = datum;
  p->next = first;
  first = p;
}

void IntList::push_front(int datum) {   // alternative version
  first = new Node{ datum, first };
}
```

IntList

empty, front, print

```
bool IntList::empty() const {
  return first == nullptr;      // or just return !first;
}
int & IntList::front() {
  assert(!empty());
  return first->datum;
}

void IntList::print(std::ostream &os) const { // no extra space
  assert(!empty());
  os << first -> datum;
  for (Node *pnode=first->next; pnode; pnode=pnode->next)
    os << " " << pnode->datum;
}
void IntList::print(std::ostream &os) const { // extra space?
  for (Node *pnode=first; pnode; pnode=pnode->next)
    os << pnode->datum << " ";  // (Mooshak) presentation error?
}
```

IntList

`pop_front` – two buggy implementations

Two wrong versions!

```
void IntList::pop_front() {
  assert(!empty());
  delete first;
  first = first->next;
}
```

Uses `first` after destroying it.

```
void IntList::pop_front() {
  assert(!empty());
  first = first->next;
  delete first;
}
```

Destroys `first` but needs it.

IntList

pop_front – two correct implementations

Correct versions:

```
void IntList::pop_front() {
  assert(!empty());
  Node *victim = first;
  first = first->next;
  delete victim;
}
```

```
void IntList::pop_front() {
  assert(!empty());
  Node *new_first = first->next;
  delete first;
  first = new_first;
}
```

IntList

Constructors, Assignment, Destructor

```
class IntList {
  ...
private:
  void pop_all();    // Removes all the elements from this list.
  void push_all(const IntList &other); // Adds all from other
};

IntList::IntList() : first(nullptr) {}  // construct empty list
IntList::IntList(const IntList &other) : IntList() {
  push_all(other);    // construct and copy values
}
IntList & IntList::operator=(const IntList &rhs) {
  if (this != &rhs) {
    pop_all();
    push_all(rhs);
  }
  return *this;
}
IntList::~IntList() {   pop_all(); }    // destructor
```

IntList

Private function members – push_all and pop_all

```
void IntList::pop_all() {
  while (!empty()) {
    pop_front();
  }
}

void IntList::push_all(const IntList &other) { //reverses order
  for (Node *node_ptr = other.first; node_ptr;
                                  node_ptr = node_ptr->next) {
    push_front(node_ptr->datum);
  }
}

void IntList::push_all(const IntList &other) { // keeps order
  for (Node *node_ptr = other.first; node_ptr;
                                 node_ptr = node_ptr->next) {
     push_back(node_ptr->datum);
  }
}
```

IntList -push_back

```
void IntList::push_back(int datum) {
  Node *new_node = new Node{ datum, nullptr };
  if (empty()) {
    first = new_node;
  } else {
    // find last node
    Node *node_ptr = first;
    for (; node_ptr->next; node_ptr = node_ptr->next);
    // set last node's next to new_node
    node_ptr->next = new_node;
  }
}
```

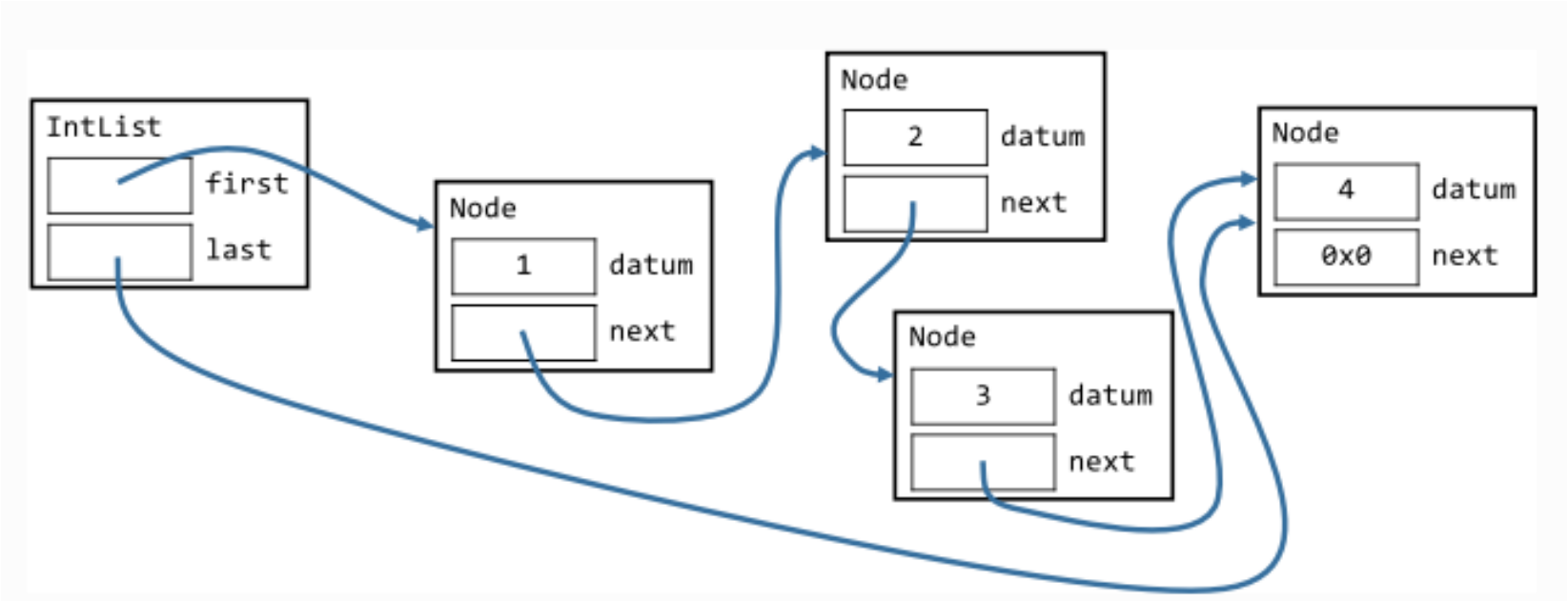
Very inefficient algorithm.

The core problem is our list implementation only keeps track of the first node.

Extension to a double-ended list

Defining a **double-ended list** (keeps track of `first` and `last` nodes):

```
class IntList {
  ...
private:
  Node *first;
  Node *last;
};
```



Extension to a double-ended list

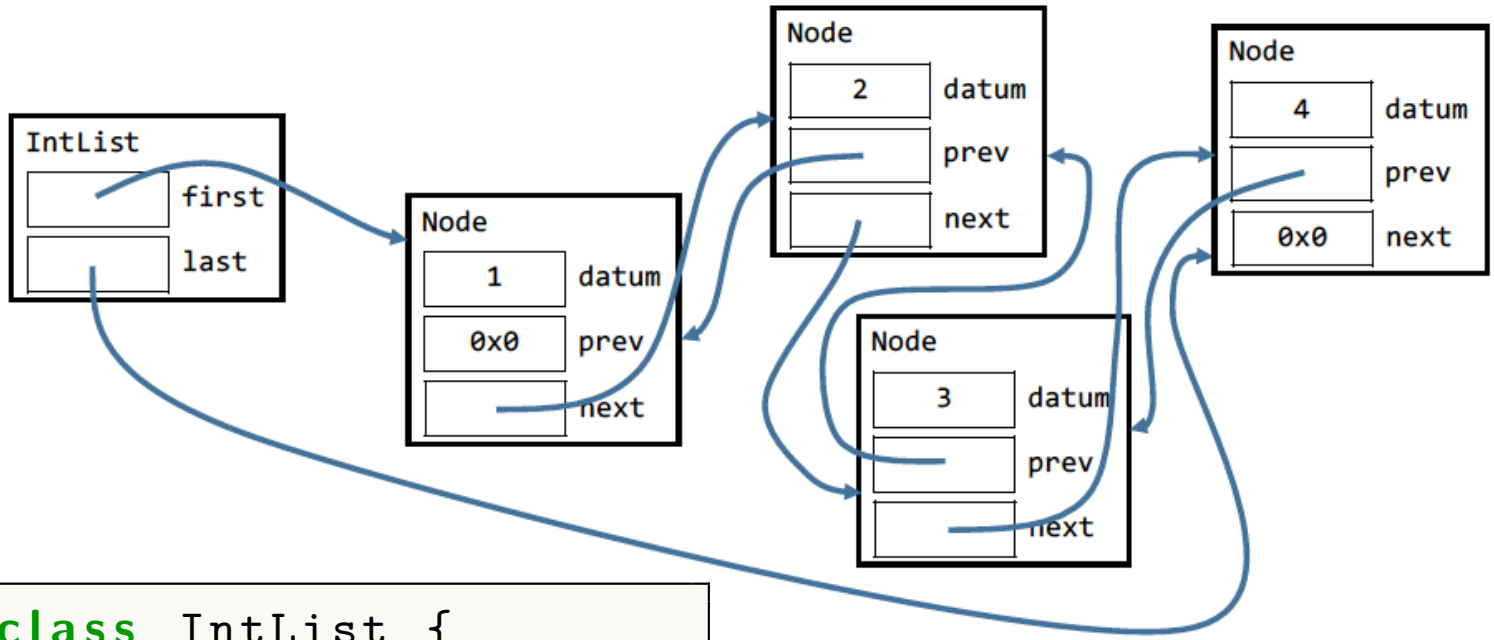
`push_back` takes $\mathcal{O}(1)$ for double-ended lists:

```
void IntList::push_back(int datum) {
  Node *new_node = new Node{ datum, nullptr };
  if (empty()) {
    first = last = new_node;
  } else {
    last = last->next = new_node;  // assigns from right to left
  }
}

// The definitions of other function members must be updated also.
```

But, what about the time complexity of `pop_back`? Still $\mathcal{O}(size)$.

Doubly-linked lists

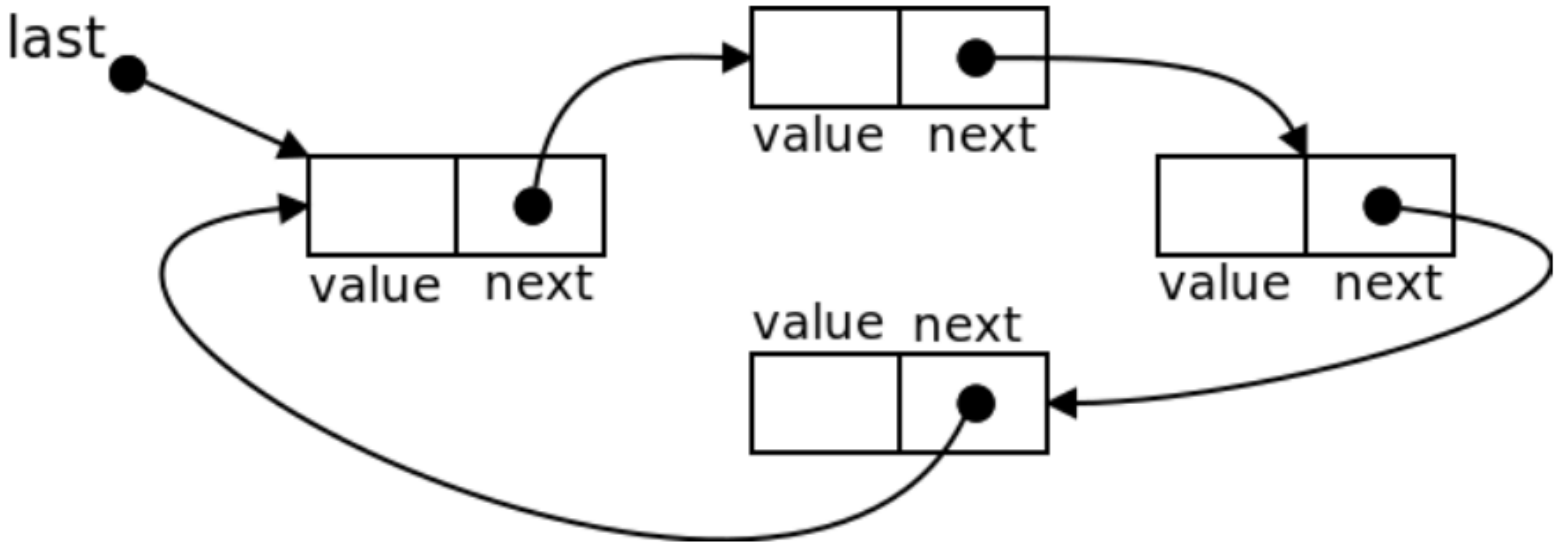



```
class IntList {
  ...
private:
  struct Node {
    int datum;
    Node *prev, *next;
  };
  Node *first, *last;
};
```

`pop_back` takes $\mathcal{O}(1)$ for doubly-linked lists.

C++ standard library provides both implementations: `std::forward_list` is a singly linked list, while `std::list` is a doubly linked list.

Circular Lists



Singly-linked circular list

And, doubly-linked circular list?

List Template

Generalizing the `IntList` class to hold objects of other types by making it a template.

```
template <typename T>          // typename or class
class List {
public:
  List();
  void empty() const;
  T & front();
  void push_front(const T &datum);
  void pop_front();
  void push_back(const T &datum);
  void pop_back();
  ...
private:
  struct Node {
    T datum;
    Node *prev, *next;
  };
  Node *first, *last;
};
```

STL: Class *list*

C++ | Containers library | **std::list**

std::list

Defined in header <list>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class list;
```
(1)

```
namespace pmr {
    template< class T >
    using list = std::list<T, std::pmr::polymorphic_allocator<T>>;
}
```
(2) (since C++17)

`std::list` is a container that supports constant time insertion and removal of elements from anywhere in the container. Fast random access is not supported. It is usually implemented as a doubly-linked list. Compared to `std::forward_list` this container provides bidirectional iteration capability while being less space efficient.

Adding, removing and moving the elements within the list or across several lists does not invalidate the iterators or references. An iterator is invalidated only when the corresponding element is deleted.

`std::list` meets the requirements of *Container*, *AllocatorAwareContainer*, *SequenceContainer* and *ReversibleContainer*.

https://en.cppreference.com/w/cpp/container/list

(constant time insertion and removal only if its position (iterator) is given)

Some methods of STL class *list*

```
iterator begin();
iterator end();
size_type size() const noexcept;
bool empty() const;
reference front();
reference back();
iterator insert iterator p, const T & e );
        // inserts e before p, returns iterator to e
void push_back(const T & e);
void push_front(const T & e);
iterator erase(iterator p) ;
      // erase returns iterator to the node that follows node p
void pop_front();
void pop_back();
void clear();
void sort();          // special sort method for the class list
```

Why a special sort? STL sort algorithm works with Random Access Iterators and not Bidirectional Iterators; class *list* uses Bidirectional Iterators.

STL: Class *list*

Excerpt from https://cplusplus.com/reference/list/list/insert/

```
#include <iostream>
#include <list>

int main () {
  std::list<int> mylist;
  std::list<int>::iterator it;
  for (int i=1; i<=5; ++i) mylist.push_back(i); // 1 2 3 4 5
  it = mylist.begin();
  ++it;         // it points now to number 2
  mylist.insert(it,10);     // 1 10 2 3 4 5
  // "it" still points to number 2
  mylist.insert(it,2,20);         // 1 10 20 20 2 3 4 5
  --it;         // it points now to the second 20
  std::cout << "mylist contains:";
  for (it=mylist.begin(); it!=mylist.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
```