List

CSE 2020 Computer Science II

Learning Objectives

- Explain abstract data type (ADT) and data structure, define List ADT
- implement List ADT using array, including insertion, deletion, and search operations.
- implement linked list, including insertion, deletion, and search operations.
- implement doubly linked list, including insertion, deletion, and search operations.
- apply list class and vector class defined in STL

Abstract Data Types

- Abstract data type (ADT) represents a set of data items together with a set of behaviors or operations that can manipulate data items.
  - shows an abstract and logical form of a data type
  - is a high-level abstract description and logical picture of the defined data type
  - describes what this data type is, what operations the data type supports
  - in C++, class definition file, .h

Data Structures

- Data structure is the implementation of an abstract data type.
  - is a concrete and physical form of a data type
  - focuses on implementation of operations supported by the data type
  - shows the implementation of the defined ADT based on how data items are stored/organized in memory
  - an ADT can be implemented using different data structures
  - in C++, class implementation file, .cpp. Please note, the implementation and definition can be combined in .cpp file.

List ADT

- A list stores a collection of elements in a linear order, that is, with the form

$$A_0, A_1, A_2, A_3, \ldots, A_i, A_{i+1}, \ldots, A_{n-2}, A_{n-1}$$

- The size of the list is n, an empty list with n = 0
- For any non-empty list,
  - the first element is A_0,
  - the last element is A_{n-1},
  - A_{i+1} follows A_i, A_i proceeds A_{i+1}
  - every element has a precedent element, except A_0
  - every element has a succeeding element, except A_{n-1}

Operations of List ADT

The popular operations are:

- int get_size() const: return the number of elements in the list
- bool empty() const: return true if the list is empty
- void clear(): remove all elements in the list
- void push(const T & x): add x to the list
  - push_back(const T & x): add x to the back
  - push_front(const T & x): add x to the front
- void pop(): remove an element
  - pop_back(): remove back element
  - pop_front(): remove front element
- bool find(const T& item) const: return true if item is in the list
- void remove(const T & x): remove x from list

List class template

```
template <typename T>
class List {
    public:
      List();
      ~List();
      bool empty( ) const;
      void clear();
      int get_size() const;
      void push_front(const T& item);
      void pop_front();
      void push_back(const T& item);
      void pop_back();
      bool find(const T& item) const;
      void remove(const T& item);
    private: ….
};
```

Implementation of List

- Using Array implement List ADT
- Using linked structure implement List ADT
  - Linked list
  - Doubly linked list

Iterator in List

- Iterator represents a position in the list
- iterator is a nested class
  - private attribute pointer *current* points to the current node/element (the address of current node/element)
  - operations
    - dereference * returns the element of current node/current element
    - prefix ++ returns the address of next node/element
    - ==, != return true if the address passed is same (different) to the address of current node/element
- In List class
  - iterator begin() returns the iterator representing the address of 1st node/element
  - iterator end() returns the address **after** the last node/element

Use Iterator

- Print the elements in a List

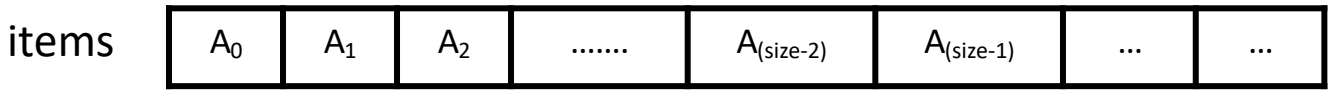
```
List<int> mylist;
....
for (List<int>::iterator itr = mylist.begin();
      itr != mylist.end(); ++itr)
       cout << *itr << “, ”;
```

- Non-member function print()

```
template <typename C>
void print(List<C> & l){
   for (typename List<C>::iterator itr = l.begin();
        itr != l.end(); ++itr)
        cout << *itr << ",";
}
```

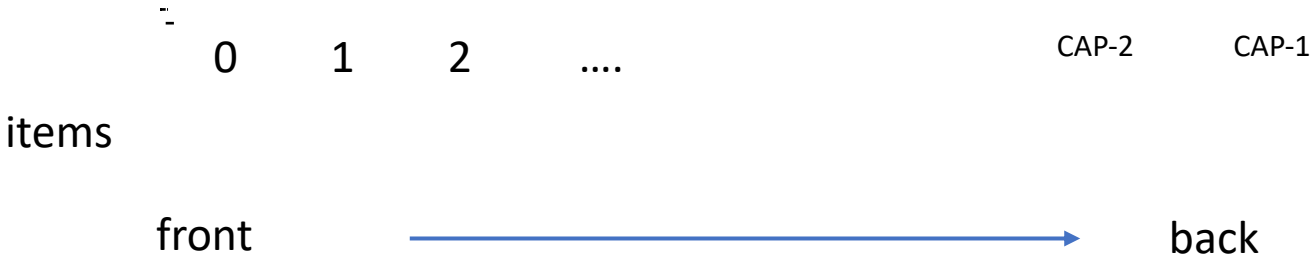
Array Implementation of List

- `ArrayList` class template is defined in ArrayList.cpp (in file ArrayList.txt)
  - private pointer *`items`* points to a dynamic array which store a collection of elements
  - private attribute *`size`* is the number of useful elements stored in the array, empty when size is 0
  - private attribute *`capacity`* is the maximal number of elements that the array can hold



Array Implementation of List Operations

- Empty list, *size* is 0



- push_back(const T & x), O(1)

```
items[size] = x;
size++;
```

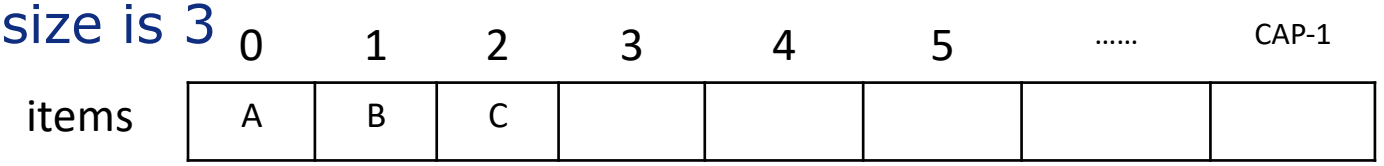
- pop_back(), O(1)

```
size--;
```

- find(const T& x): sequential search O(n)
- remove(const T& x): find x and remove x O(n)

Array Implementation of List Example

- al.push_back('A'); al.push_back('B'); al.push_back('C');
  - size is 3



- al.pop_back(); size is 2

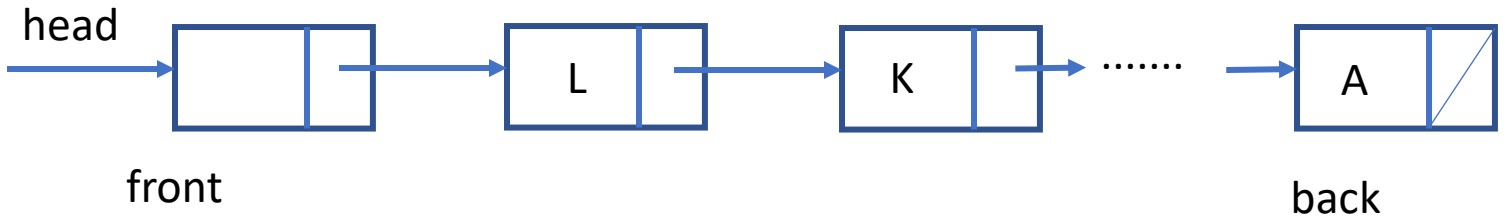
- How about `push_front()` and `pop_front()`?
- find(x) and remove(x)?

Array Implementation of List

- Use class template to define list in a general way, the type of array element is `T`, which is bound to an actual data type in main() function, as shown in TestArrayList.cpp (in file ArrayList.txt)
  - `#include “ArrayList.cpp”`
  - `ArrayList<int>`
  - `ArrayList<double>`
  - `ArrayList<string>`
  - `ArrayList<Employee>`
  - `ArrayList<Student>`

Singly Linked List

- `LinkedList` class template is defined in LinkedList.cpp (lab exercise)
  - private structure *`NodeType`* defines the node structure, contain *`data`* of type `T` and *`next`* pointer to next node
  - private pointer *`head`* points to the header node of a collection of elements, empty when *head->next* is *`nullptr`*
  - private attribute *`size`* is the number of elements stored in the linked list



Singly Linked List Operations

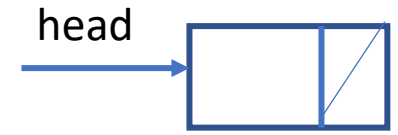
- Empty linked list, size is 0
- push_front(const T & x), O(1)

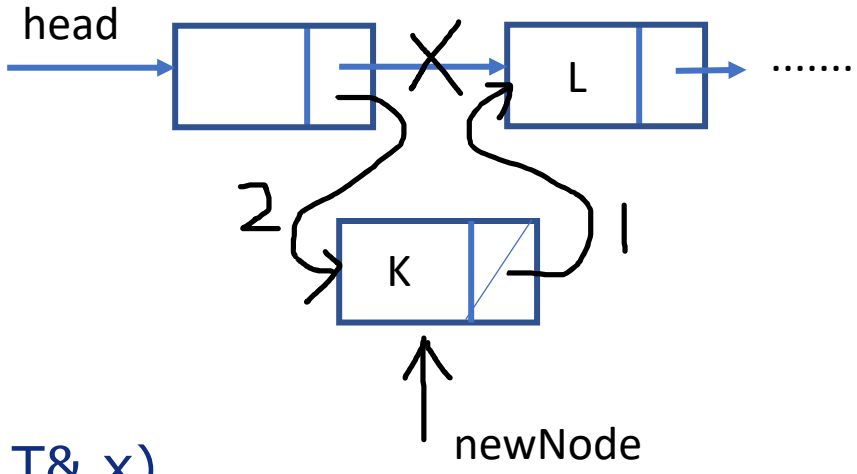
```
NodeType* newNode = new NodeType(x);
newNode->next = head->next;
head->next = newNode;
size++;
```

- pop_front(), O(1)

```
NodeType* ptr = head->next;
head->next = ptr->next;
delete ptr;
size--;
```

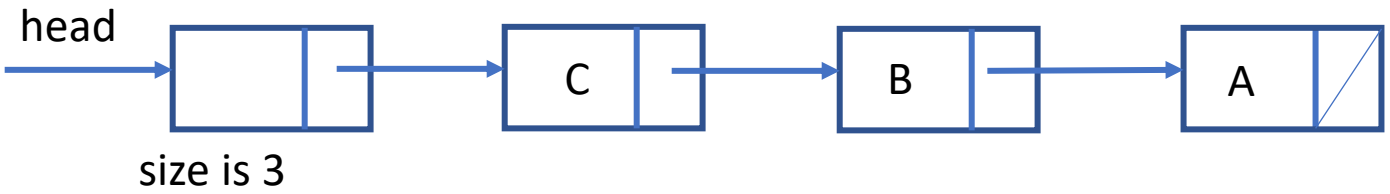
- find(const T& x), remove(const T& x)
  - sequential access each node, O(n)



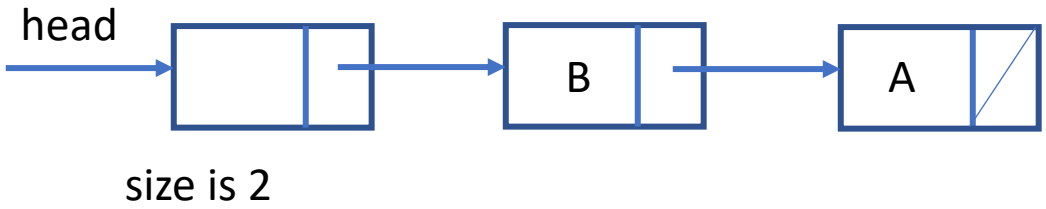


Singly Linked List Example

- ll.push_front('A'); ll.push_front('B'); ll.push_front('C');



- ll.pop_front();



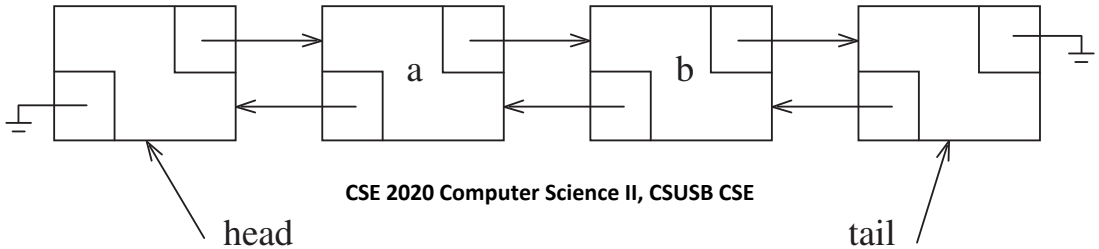
- How about push_back() and pop_back()?
- find(x) and remove(x)?

Singly Linked List

- Use class template to define list in a general way, the type of element is `T`, which is bound to an actual data type in main() function.
  - `#include` “LinkedList.cpp”
  - `LinkedList<int>`
  - `LinkedList<double>`
  - `LinkedList<string>`
  - `LinkedList<Employee>`
  - `LinkedList<Student>`

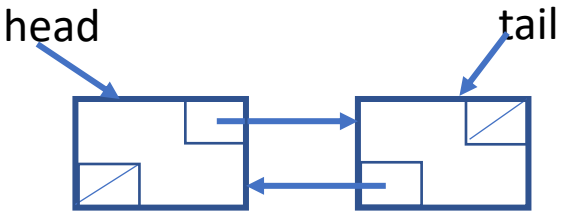
Doubly Linked List

- `DoublyLinkedList` class template is defined in DoublyLinkedList.cpp (in file DoublyLinkedList.txt)
  - private structure *`NodeType`* defines the node structure, contain *`data`* of type `T`, *`next`* pointer points to next node, *`prev`* pointer points to previous node
  - private pointer *`head`* points to the header node of a collection of elements
  - private pointer *`tail`* points to the tail node of a collection of elements
  - private attribute *`size`* is the number of elements stored in the linked list



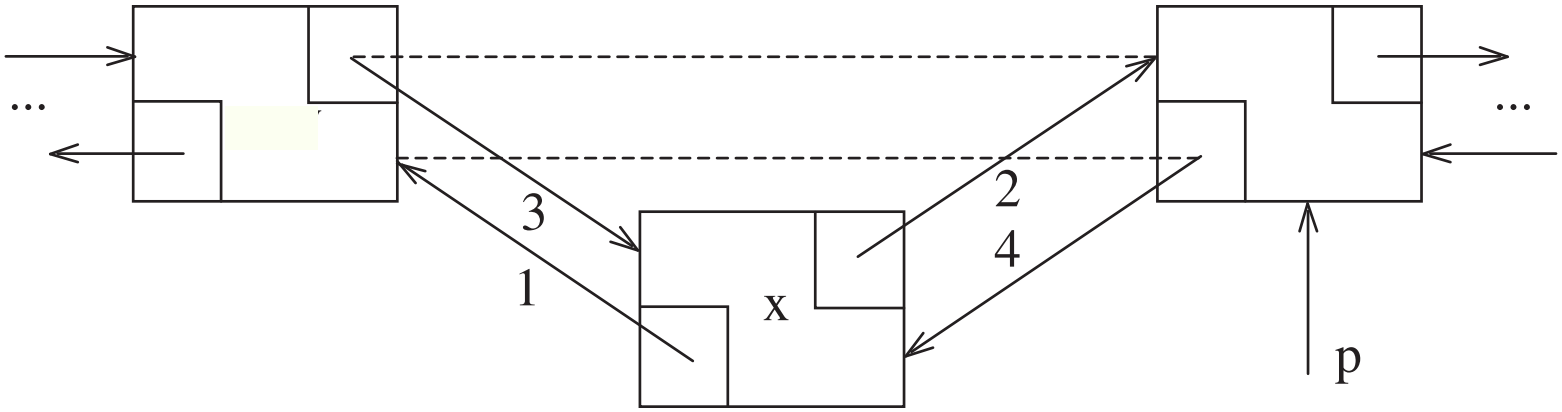
Doubly Linked List Operations

- Empty list



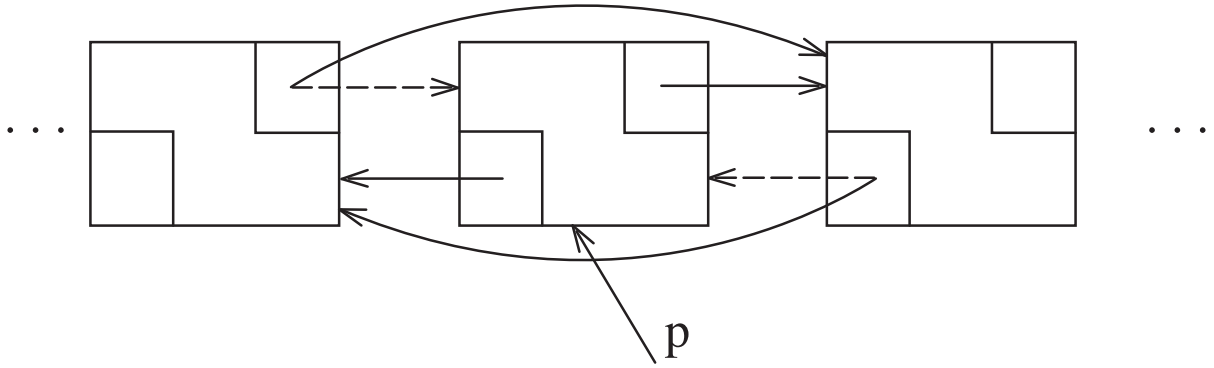
- push_front(), O(1)
- push_back(), O(1)
- pop_front(), O(1)
- pop_back(), O(1)
- find(x) and remove(x), O(n)

Insert Node in Doubly Linked List




```
NodeType* newNode = new NodeType(x);
newNode->prev = p->prev;
newNode->next = p;
newNode->prev->next = newNode;
p->prev = newNode;
```

Delete Node in Doubly Linked List




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

Doubly Linked List

- Use class template to define list in a general way, the type of element is `T`, which is bound to an actual data type in main() function, as shown in TestDoublyLinkedList.cpp (in file DoublyLinkedList.txt)
  - `#include` “DoublyLinkedList.cpp”
  - DoublyLinkedList `<int>`
  - DoublyLinkedList `<double>`
  - DoublyLinkedList `<string>`
  - DoublyLinkedList `<Employee>`
  - DoublyLinkedList `<Student>`

Array vs Linked List

- Array implementation of list has fixed capacity
- Linked list has no fixed capacity

Operations	Array Imp	Singly linked	Doubly linked
push_front(e)	O(n)	O(1)	O(1)
push_back(e)	O(1)	O(n)	O(1)
pop_front()	O(n)	O(1)	O(1)
pop_back()	O(1)	O(n)	O(1)
find(e)	O(n)	O(n)	O(n)
remove(e)	O(n) + O(n)	O(n) + O(1)	O(n)+O(1)
clear()	O(1)	O(n)	O(n)

STL vector

- In C++, Standard Template Library contains the implementation of common data structures.
- *vector* provides a growable array implementation of the List ADT

```
#include<vector>
vector<int> intv;
intv.push_back(10);
intv.pop_back();
vector<double> dblv(10);
vector<string> strv;
```

- Please review TestListVectorSTL.cpp

STL list

- *list* provides doubly linked list implementation of the List ADT

```
#include<list>
list<int> intl;
intl.push_back(10);
intl.pop_back();
list<double> dbll;
list<string> strl;
```

- Please review TestListVectorSTL.cpp