

Data Structures and Algorithms

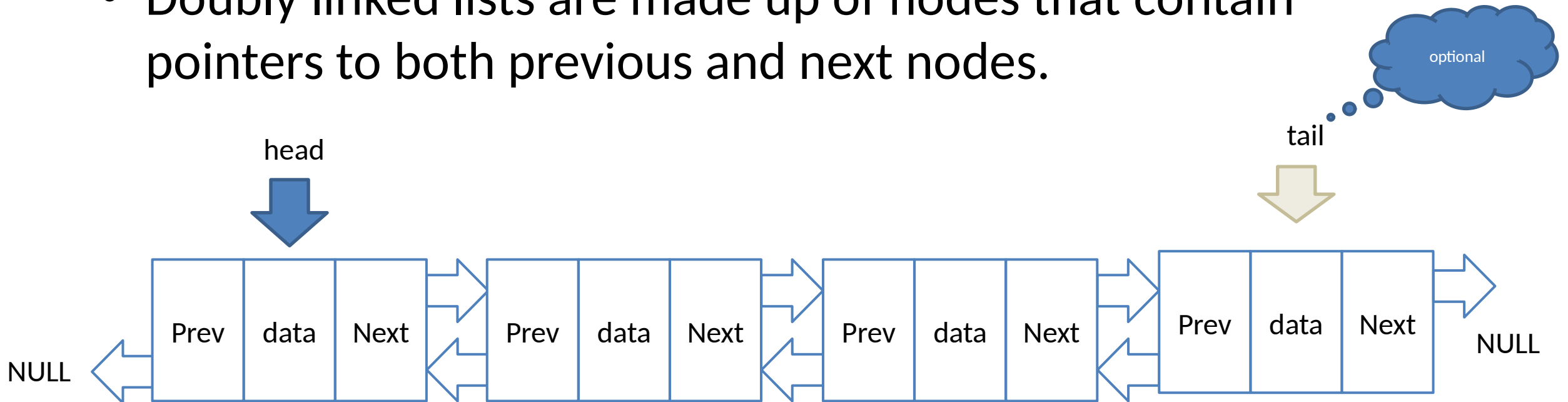
List ADT implementations:

Doubly Linked List

Circular Linked List

Doubly Linked List

- Doubly linked lists are made up of nodes that contain pointers to both previous and next nodes.



Node structure

- The nodes that make up the doubly linked list have links to both the previous and next nodes.

Pointer to previous node

Pointer to next node



```
template<typename T>
class Node {
private:
    T data;
    Node *next, *prev;
public:
    Node(const T &data, Node<T> *next = NULL,
         Node<T> *prev = NULL) {
        this->data = data;
        this->next = next;
        this->prev = prev;
    }
    template<typename U> friend class
    LinkedList;
};
```

Doubly Linked List Class

```
template<typename T>
class LinkedList {
private:
    Node<T> *head, *tail;
    int nItems; //number of items

public:
    LinkedList() {
        head = NULL;
        tail = NULL;
        nItems = 0;
    }

    bool isEmpty() {
        return head == NULL;
    }
    ...
}
```

- A pointer (head) must be defined that points to the first element of the list, as in the singly linked list.
- In addition, a pointer to the last element (tail) speeds up insertion and deletion.
- Also, defining a variable to hold the number of elements will speed things up.
- These variables can also be used in a singly linked list.
- For example, adding a tail to a singly linked list will speed up adding and reading the last element.

Add new item to the front

```
void push_front(const T& data) {  
    Node<T> *node = new Node<T>(data);  
    if (head == NULL) {  
        head = node;  
        tail = node;  
    } else {  
        head->prev = node;  
        node->next = head;  
        head = node;  
    }  
    nItems++;  
}
```

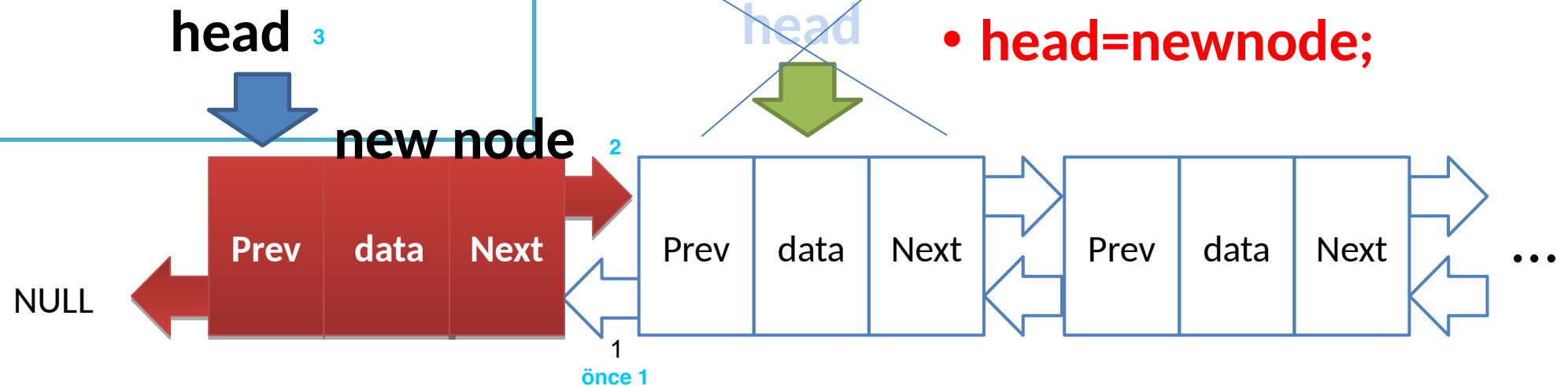
If we add elements while the list is empty, the first element and the last element are the same

If there is an element in the list, add the new element before the first element and make the first element (head) the new element.

newnode->next=head;

• head->prev=newnode;

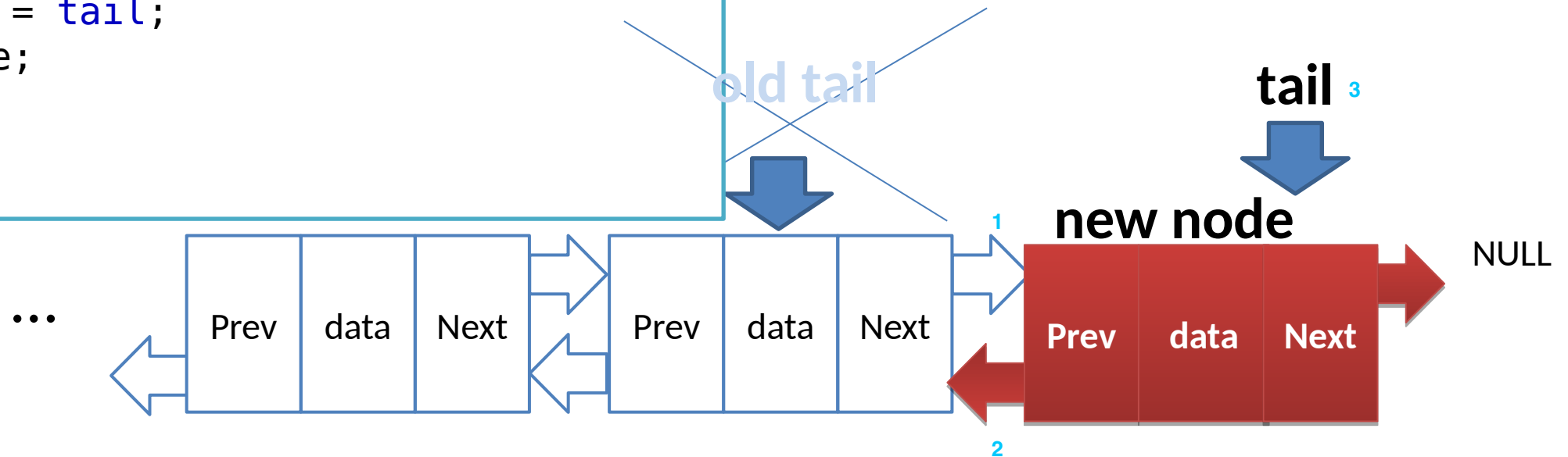
• head=newnode;



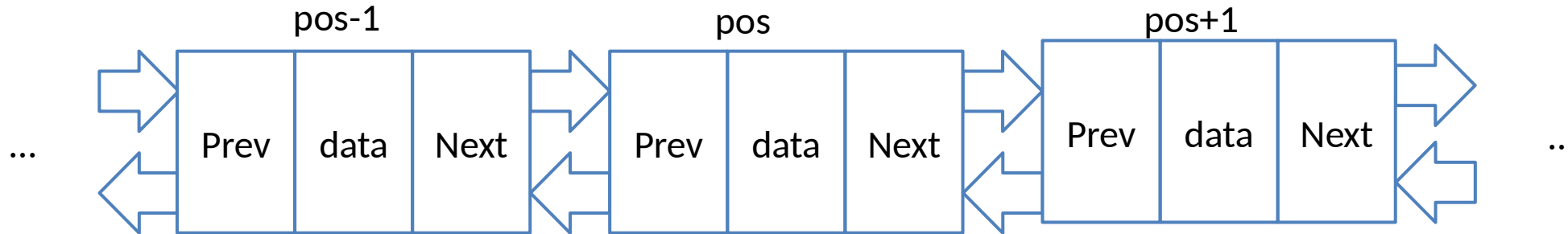
Add new item to the back

```
void push_back(const T& data) {  
    Node<T> *node = new Node<T>(data);  
    if (head == NULL) {  
        head = node;  
        tail = node;  
    } else {  
        //we can use tail to add back in O(1)  
        time  
        tail->next = node;  
        node->prev = tail;  
        tail = node;  
    }  
    nItems++;  
}
```

- Adding an element to the end is done in a similar way to adding an element to the beginning.
- However, if no end pointer (tail) is defined, it will loop through the first element until the last element is found, before adding the new node.

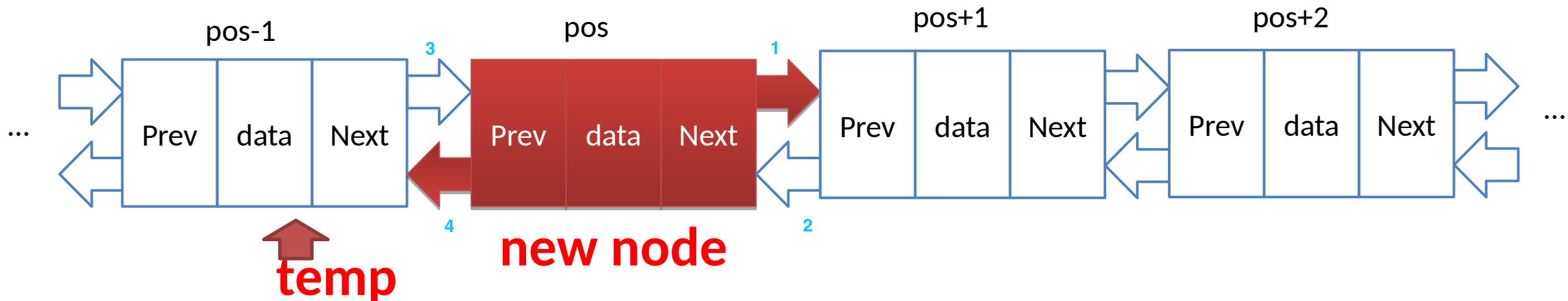


Insert an item



- **`newnode->next = temp->next;`**
- **`temp->next->prev = newnode;`**
- **`temp->next = newnode;`**
- **`newnode->prev = temp;`**

As in a singly linked list, when adding a node to the specified location, the previous (or the next, since it is doubly linked) node is found and connections are made.



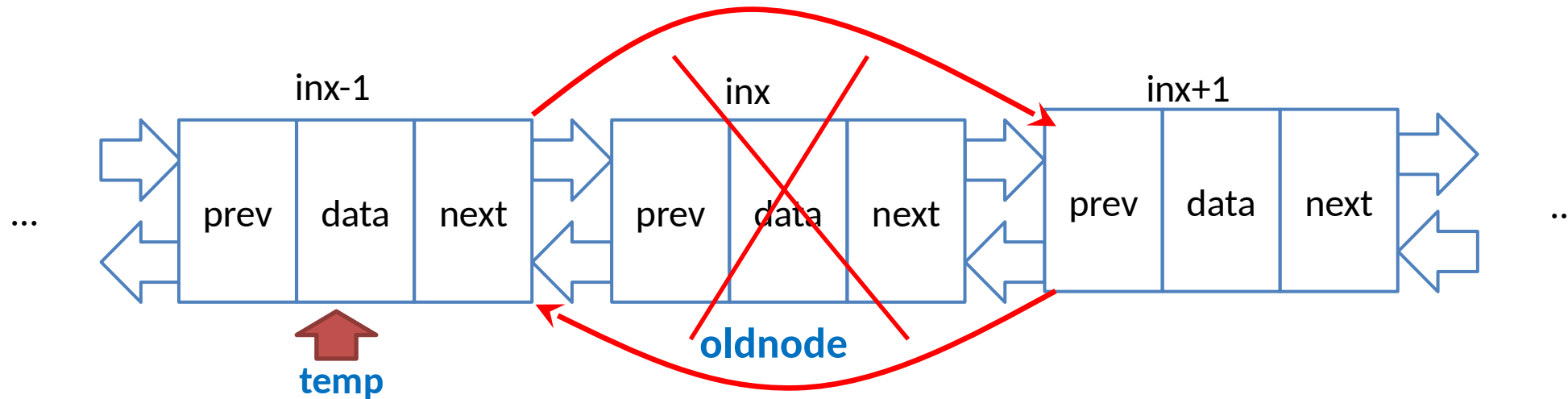
Insert an item

```
void insert(int inx, const T &data)
{
    if ((inx < 0) | (inx > nItems))
        throw IndexException();

    if (inx == 0)
        push_front(data);
    else if (inx == nItems)
        push_back(data);
    else {
        Node<T> *node = new
        Node<T>(data);
        Node<T> *temp = find_prev(inx);
        node->next = temp->next;
        temp->next->prev = node;
        temp->next = node;
        node->prev = temp;
        nItems++;
    }
}
```

- If the position is invalid, the function is terminated by throwing an error.
- If position=0 then the element is added to the beginning of the list. Here push_front() is called.
- If position>0, after a new node is created, the node at the position before the node is found and connections are made as described in the previous slide.

Remove an item



- **oldnode = temp->next;**
- **temp->next = oldnode ->next;**
- //if it is not the last element
- **if (temp->next != NULL)**
- **oldnode ->next->prev = temp;**
- //collect garbage
- **delete oldnode ;**

Remove an item

```
void remove(int inx) {  
    if ((inx < 0) | (inx >= nItems))  
        throw IndexException();  
    if (head == NULL)  
        throw EmptyListException();  
  
    Node<T> *rem;  
    if (inx == 0) {  
        rem = head;  
        head = head->next;  
        if (head != NULL)  
            head->prev = NULL;  
    } else {  
        tail = NULL;  
        Node<T> *rem;  
        Node<T> *temp = find_prev(inx);  
        if (inx == (nItems - 1))  
            tail = temp;  
        rem = temp->next;  
        temp->next = rem->next;  
        if (temp->next != NULL)  
            rem->next->prev = temp;  
    }  
  
    delete rem;  
    nItems--;  
}
```

Find previous
node

Make
connections

Read or change an item

```
T& at(int inx) {  
    if ((inx < 0) | (inx >= nItems)) throw  
    IndexException();  
    if (nItems==1) return head->data;  
    else {  
        Node<T> *temp = find_prev(inx + 1);  
        return temp->data;  
    }  
}
```

If there is only one
item, return head-
>data

Find the node
at i

Read or change an item

```
void clear() {  
    Node<T> *temp = head;  
    Node<T> *delnode;  
  
    while (temp != NULL) {  
        delnode = temp;  
        temp = temp->next;  
        delete delnode;  
    }  
  
    head = NULL;  
    tail = NULL;  
    nItems = 0;  
}
```

Delete all
nodes one by
one

Empty list
status

Using iterators

```
template <class T>
class ListIterator{
private:
Node<T> *current;
public:
ListIterator(Node<T> *current=NULL){
this->current=current;
}
void next(){
if (current==NULL) throw IndexException();
current=current->next;
}
void prev(){
if (current==NULL) throw IndexException();
current=current->prev;
}
T& get()const{
return current->data;
}
bool end(){
return (current==NULL);
}

//LinkedList class will be provided access to private
variables
template <class U> friend class LinkedList;
};
```

Initially set
pointer to a
list item

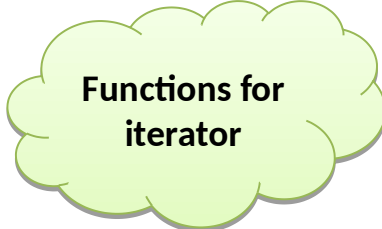
Visit next or
previous

- An iterator is an object that enables a programmer to **traverse a container**, particularly lists
- An external iterator may be thought of as a type of pointer that has two primary operations: referencing one particular element in the object collection (called element access), and modifying itself so it points to the next element (called element traversal)
- **The primary purpose of an iterator is to allow a user to process every element of a container while isolating the user from the internal structure of the container.** This allows the container to store elements in any manner it wishes while allowing the user to treat it as if it were a simple sequence or list. An iterator class is usually designed in tight coordination with the corresponding container class.
<https://en.cppreference.com/w/cpp/iterator>

Using iterators

```
#include <iostream>
#include "Exceptions.hpp"
#include "Node.hpp"
#include "ListIterator.hpp"
using namespace std;
```

```
template<typename T>
class LinkedList {
...
ListIterator<T> begin()const{
    if (head == NULL) EmptyListException();
    return ListIterator<T>(head);
}
ListIterator<T> end()const{
    if (tail == NULL) EmptyListException();
    return ListIterator<T>(tail);
}
...
}
```



Functions for
iterator

Using iterators

```
LinkedList<string> list;
list.push_front("apple");
list.push_front("orange");
list.insert(0, "potato");
list.push_back("cabbage");
list.push_back("cucumber");
list.insert(1, "carrot");
list.insert(6, "garlic");

cout<<"--print list:"<<endl;
for (ListIterator<string>
itr=list.begin();!itr.end();itr.next())
    cout<<itr.get()<<endl;

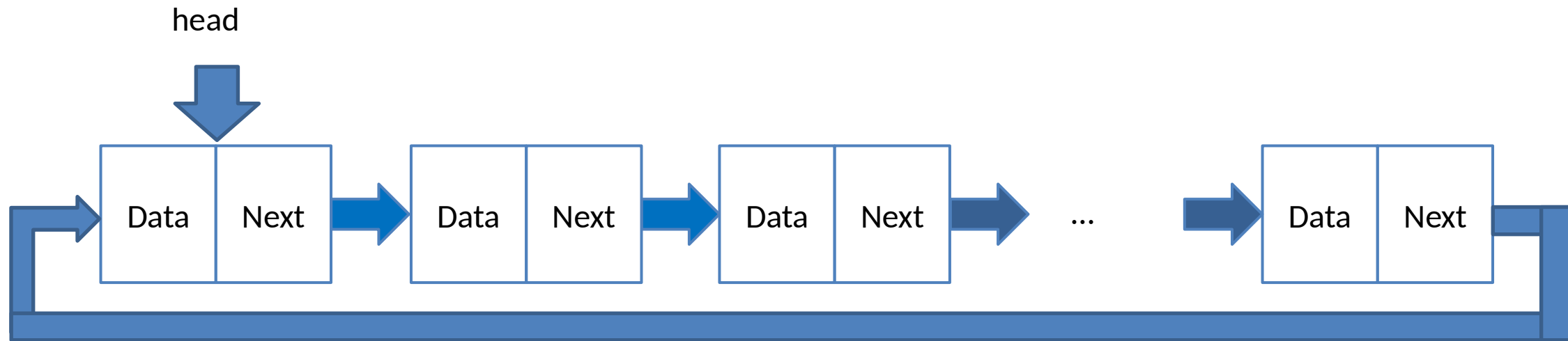
cout<<endl;
cout<<"--print list in reverse
order:"<<endl;
for (ListIterator<string> itr=list.end();!
itr.end();itr.prev())
cout<<itr.get()<<endl;
list.remove(5);
```

```
--print list:
potato
carrot
orange
apple
cabbage
cucumber
garlic

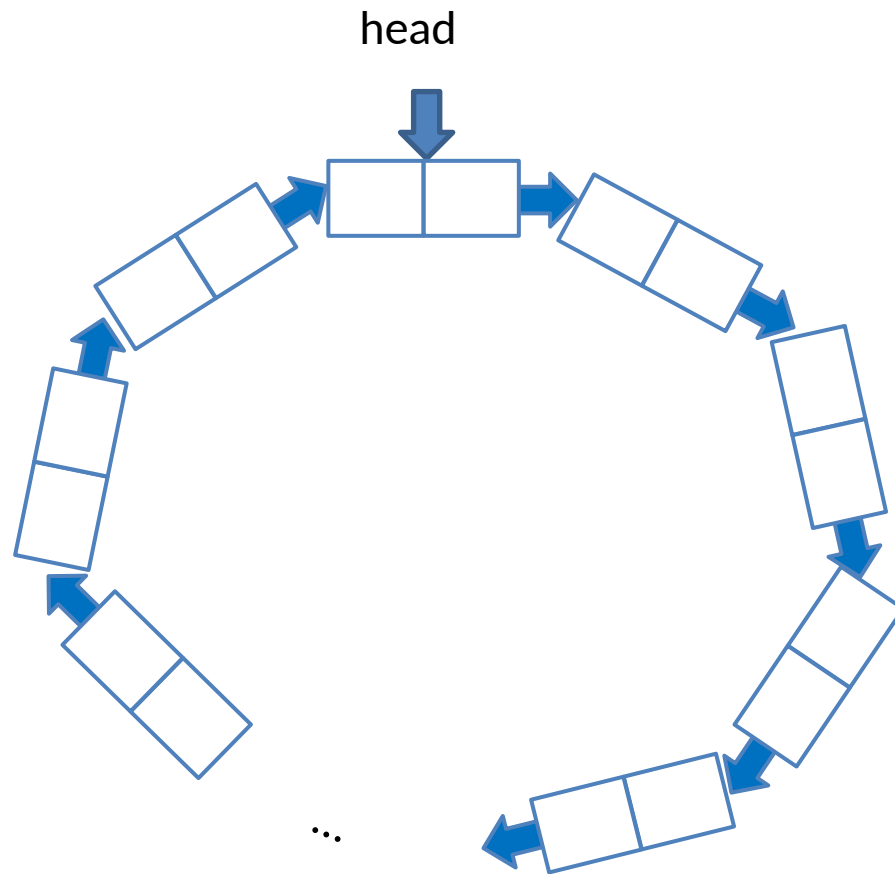
--print list in reverse order:
garlic
cucumber
cabbage
apple
orange
carrot
potato
```

Circular Linked List

- Singly Circular Linked List



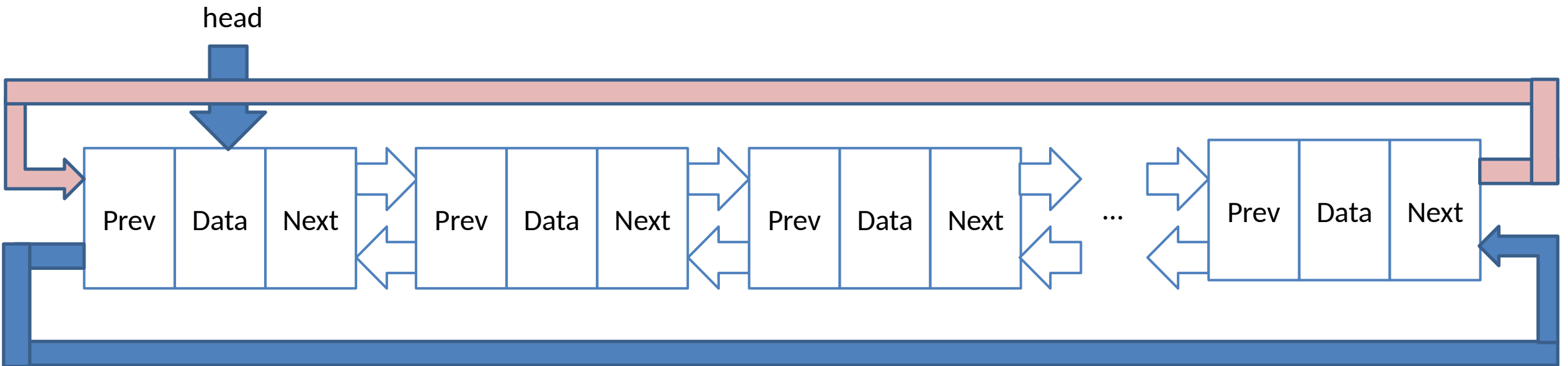
Circular Linked List



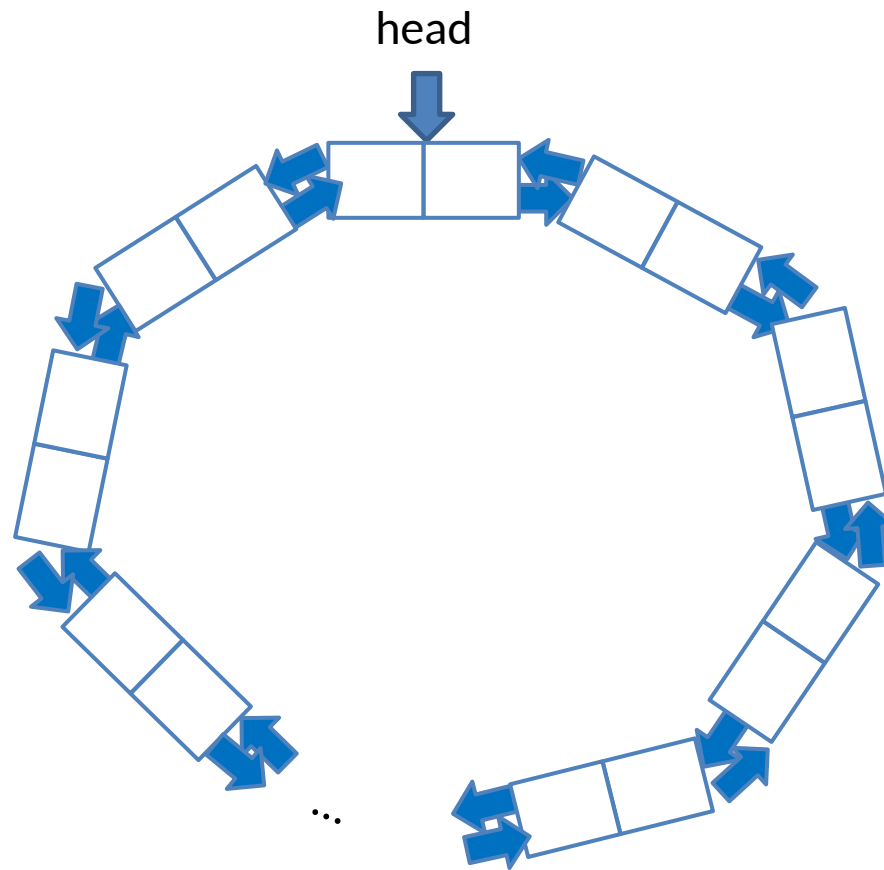
Singly
Circular
Linked List

Circular Linked List

- Doubly Circular Linked List



Circular Linked List



Doubly
Circular
Linked List

vector and list in C++ STL

std::vector

Vectors are sequence containers representing **arrays that can change in size**.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Internally, **vectors use a dynamically allocated array to store their elements**. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a **relatively expensive** task in terms of processing time, and thus, **vectors do not reallocate** each time an element is added to the container.

std::list

Lists are sequence containers that **allow constant time insert and erase operations** anywhere within the sequence, and iteration in both directions.

List containers are implemented as doubly-linked lists; Doubly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept internally by the association to each element of a link to the element preceding it and a link to the element following it.

vector and list in C++ STL

```
#include <vector>
```

```
int main(int argc, char** argv) {
```

```
    vector<string> vec;
```

```
    cout<<"vec.capacity()="<<vec.capacity()<<endl;
```

```
    vec.push_back("Pazartesi");//sona ekle
```

```
    cout<<"vec.capacity()="<<vec.capacity()<<endl;
```

```
    vec.push_back("Salı");
```

```
    cout<<"vec.capacity()="<<vec.capacity()<<endl;
```

```
    vec.push_back("Perşembe");
```

```
    cout<<"vec.capacity()="<<vec.capacity()<<endl;
```

```
    vec.push_back("Cuma");
```

```
    cout<<"vec.capacity()="<<vec.capacity()<<endl;
```

```
    vec.push_back("Pazar");
```

```
    cout<<"vec.capacity()="<<vec.capacity()<<endl;
```

```
    yazdir<string>(vec);
```

```
template <typename T>
```

```
void yazdir(vector<T> vec) {
```

```
    cout<<"\n---liste-----"<<endl;
```

```
    for(vector<string>::iterator itr=vec.begin();
```

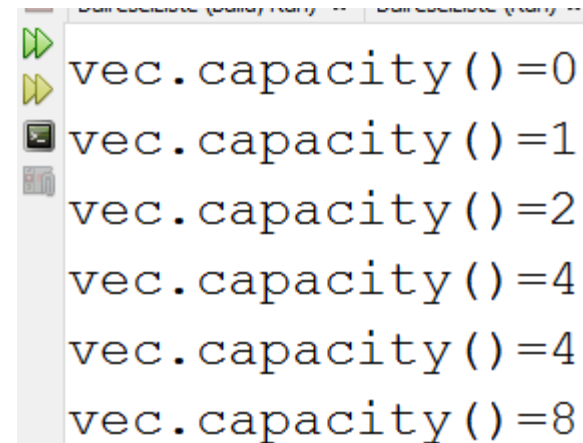
```
        itr != vec.end();
```

```
        ++itr) {
```

```
        cout<<" " <<*itr<<endl;
```

```
    }
```

```
}
```



```
vec.capacity()=0
vec.capacity()=1
vec.capacity()=2
vec.capacity()=4
vec.capacity()=4
vec.capacity()=8
```

vector and list in C++ STL

```
vec.pop_back(); //sondan sil
yazdir<string>(vec);

//ilk elemanı sil
vec.erase(vec.begin());
yazdir<string>(vec);

// 3. konumdaki elemanı sil
vec.erase(vec.begin()+2);
yazdir<string>(vec);

// 3. konuma eleman ekle
vec.insert(vec.begin()+2, "Çarşamba");
yazdir<string>(vec);

//0. konumdaki elemanı değiştir
cout<<"\nvec.at(0)="<<vec.at(0)<<endl;
vec.at(0)="Pazartesi";
cout<<"vec.at(0)="<<vec.at(0)<<endl;
```

vector and list in C++ STL

```
#include <list>
```

```
list<string> list1;

list1.push_back("Pazartesi"); //sona ekle
list1.push_back("Salı");
list1.push_back("Çarşamba");
list1.push_back("Cuma");
yazdir<string>(list1);
```

```
list<string>::iterator itr=list1.begin();
list1.insert(itr,"Perşembe");
itr++;
list1.insert(itr,"Cumartesi");
yazdir<string>(list1);
```

```
template <typename T>
void yazdir(list<T> list1){
    cout<<"\n---liste-----"<<endl;
    for(list<string>::iterator itr=list1.begin();
        itr != list1.end();
        ++itr){
        cout<<" "<<*itr<<endl;
    }
}
```

Advantages of Linked List

- A linked list is a dynamic data structure that can grow and shrink in size at runtime by allocating and deallocating memory. As a result, there is no need to specify the linked list's initial size.
- A linked list is frequently used to build linear data structures such as stacks and queues.
- The linked list makes it much easier to insert and delete items. After an element is inserted or deleted, there is no need to move it; only the address in the next pointer needs to be updated.

Disadvantages of Linked List

- A linked list requires more memory than an array. Because a pointer is necessary to store the address of the next entry in a linked list, it consumes additional memory.
- Traversing a linked list takes longer than traversing an array. A linked list, unlike an array by index, does not provide direct access to an entry. To get to a node at position n , for example, you have to go through all the nodes before it.
- Reverse traversing is not possible in a singly linked list, but it is possible in a doubly-linked list since each node carries a pointer to the previously connected nodes. Extra memory is necessary for the back pointer to execute this, resulting in memory waste.
- Because of the dynamic memory allocation in a linked list, random access is not possible.

<https://www.interviewbit.com/linked-list-interview-questions/>