Linked Lists

Linked List Basics

Linked lists and arrays are similar since they both store collections of data.

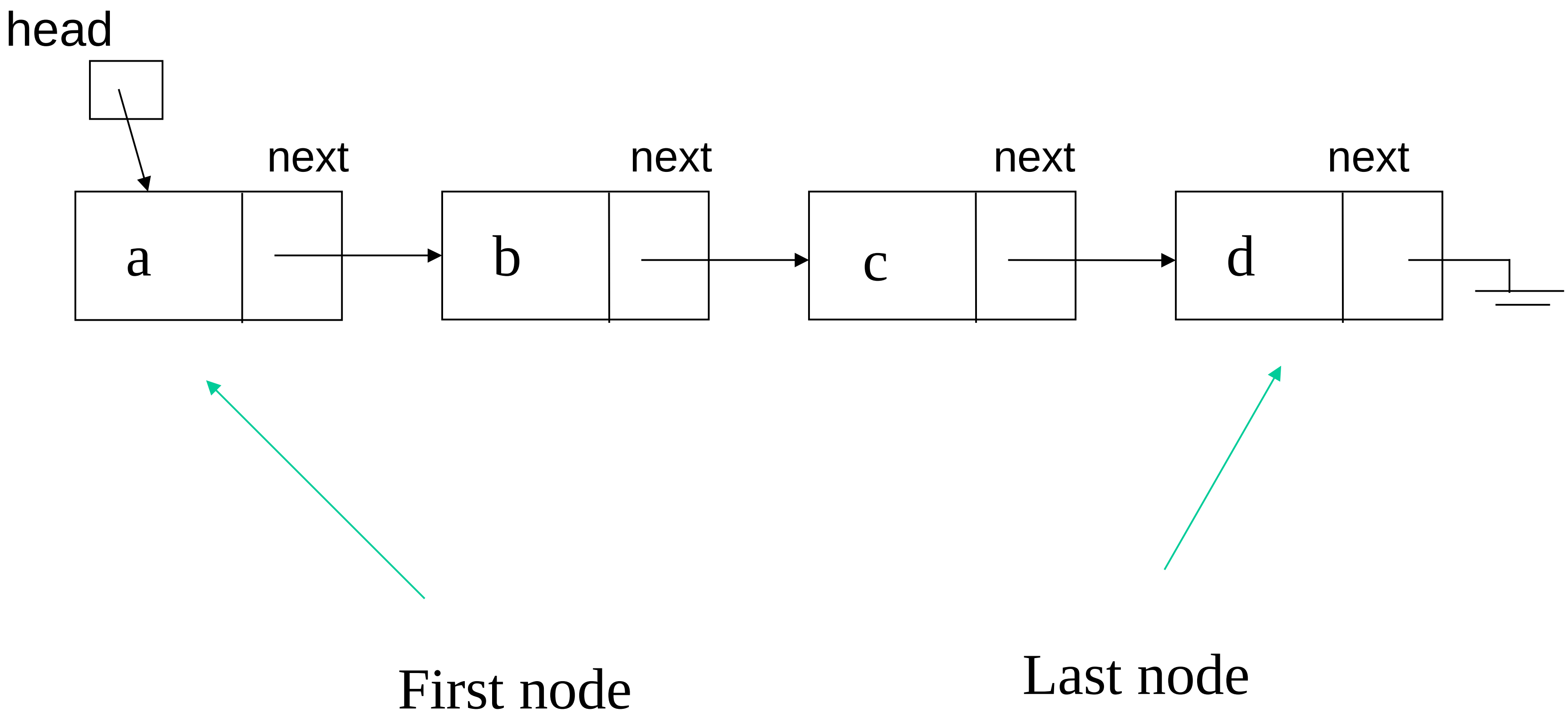
The *array's* features all follow from its strategy of allocating the memory for all its elements in one block of memory.

Linked lists use an entirely different strategy: linked lists allocate memory for each element separately and only when necessary.

Linked List Basics

- Linked lists are used to store a collection of information (like arrays)
- A linked list is made of nodes that are pointing to each other
- We only know the address of the first node (head)
- Other nodes are reached by following the “next” pointers
- The last node points to NULL

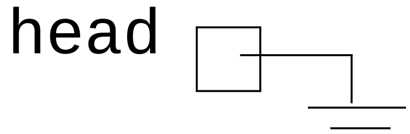
Linked Lists



Empty List

Empty Linked list is a single pointer having the value NULL.

```
head = NULL;
```



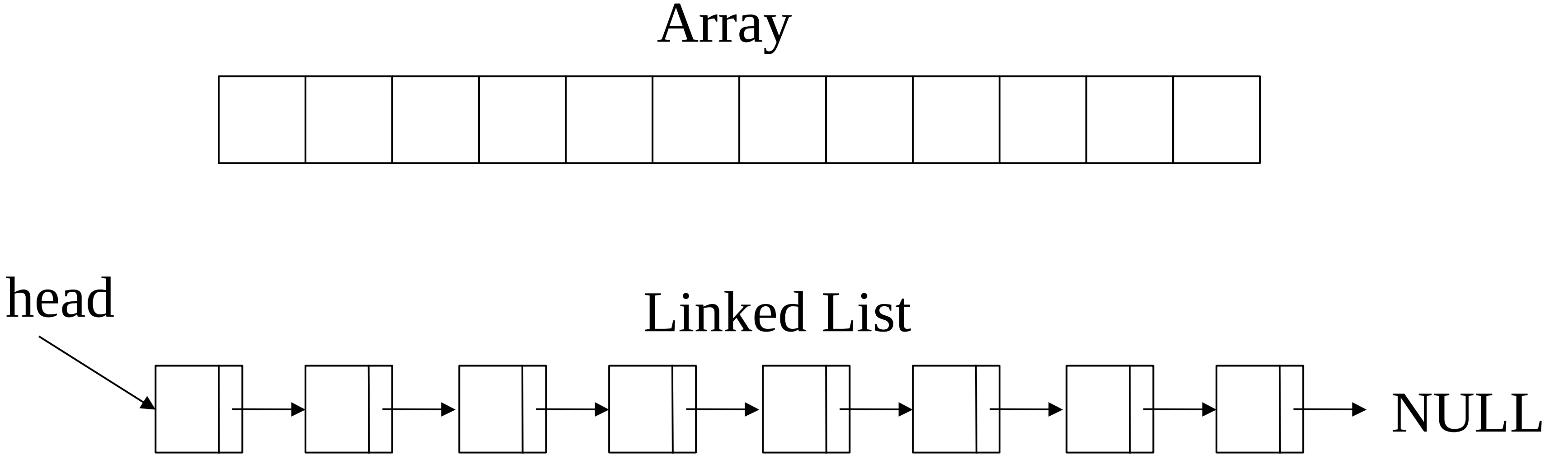
Linked List Basics

- Each node has (at least) two fields:
  - Data
  - Pointer to the next node

data	ptr

Linked List vs. Array

- In a linked list, nodes are not necessarily contiguous in memory (each node is allocated with a separate "new" call)
- Compare this to arrays which are contiguous



Linked List vs. Array

- **Advantages of Arrays**
  - Can directly select any element
  - No memory wasted for storing pointers
- **Disadvantages of Arrays:**
  - Fixed size (cannot grow or shrink dynamically)
  - Need to shift elements to insert an element to the middle
  - Memory wasted due to unused elements
- **Advantages of Linked Lists:**
  - Dynamic size (can grow and shrink as needed)
  - No need to shift elements to insert into the middle
  - Size can exactly match the number of elements (no wasted memory)
- **Disadvantages of Linked Lists**
  - Cannot directly select any element (need to follow ptrs)
  - Extra memory usage for storing pointers

Linked List vs. Array

- In general, we use linked lists if:
  - The number of elements that will be stored cannot be predicted at compile time
  - Elements may be inserted in the middle or deleted from the middle
  - We are less likely to make random access into the data structure (because random access is expensive for linked lists)

Linked List Implementation

- A linked list node can be represented as follows:

```
template <class T>
class Node {
public:
     T element;
   Node *next;
};
```

Linked List Node Implementation

- We can add a constructor to simplify creating a new node:

```
template <class T>
class Node {
public:
  Node(const T& e = T(), Node *n = NULL) :
      element(e), next(n) { }

  T element;
  Node *next;
};
```

Linked List Implementation

- A linked list can be defined as follows
- Note that the constructor creates an empty list by making the head point to NULL

```
template <class T>
class List {
private:
  Node<T> *head;
public:
  List() : head(NULL) {}
};
```

Basic Linked List Operations

- Insert a node
- Delete a node
- List Traversal
- Searching a node
- Is Empty

Linked List Operations

- **isEmpty():** returns true if the list is empty, false otherwise

```
template <class T>
class List {
private:
  Node<T> *head;
public:
  List() : head(NULL) {}
  bool isEmpty() const;
};
```

Linked List Operations

- **isEmpty():** returns true if the list is empty, false otherwise

```
template <class T>
bool List<T>::isEmpty() const {
      return head == NULL;
}
```

Linked List Operations

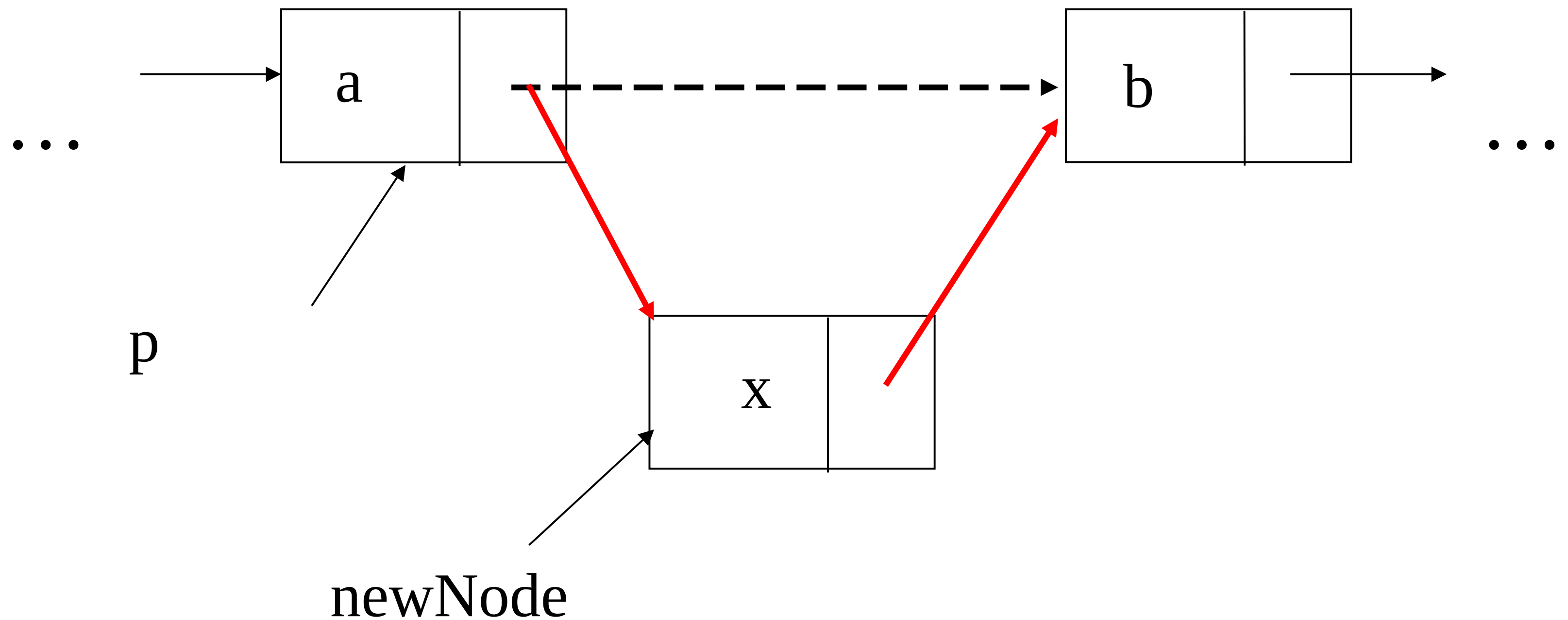
- **first():** returns the first node of the list

```
template <class T>
class List {
private:
  Node<T> *head;
public:
  List() : head(NULL) {}
  bool isEmpty() const;
  Node<T>* first();
};
```

Linked List Operations

```
template <class T>
Node<T>* List<T>::first() {
    return head;
}
```

Insertion in a linked list



Insertion

- For insertion, we have to consider two cases:
  1. Insertion to the middle
  2. Insertion before the head (or to an empty list)
- In the second case, we have to update the head pointer as well

Linked List Operations : insert

- **insert(const T& data, Node<T>* p):** inserts a new element containing data after node p

```
template <class T>
class List {
private:
  Node<T> *head;
public:
  ...
  void insert(const T& data, Node<T>* p);
  ...
};
```

Linked List Operations

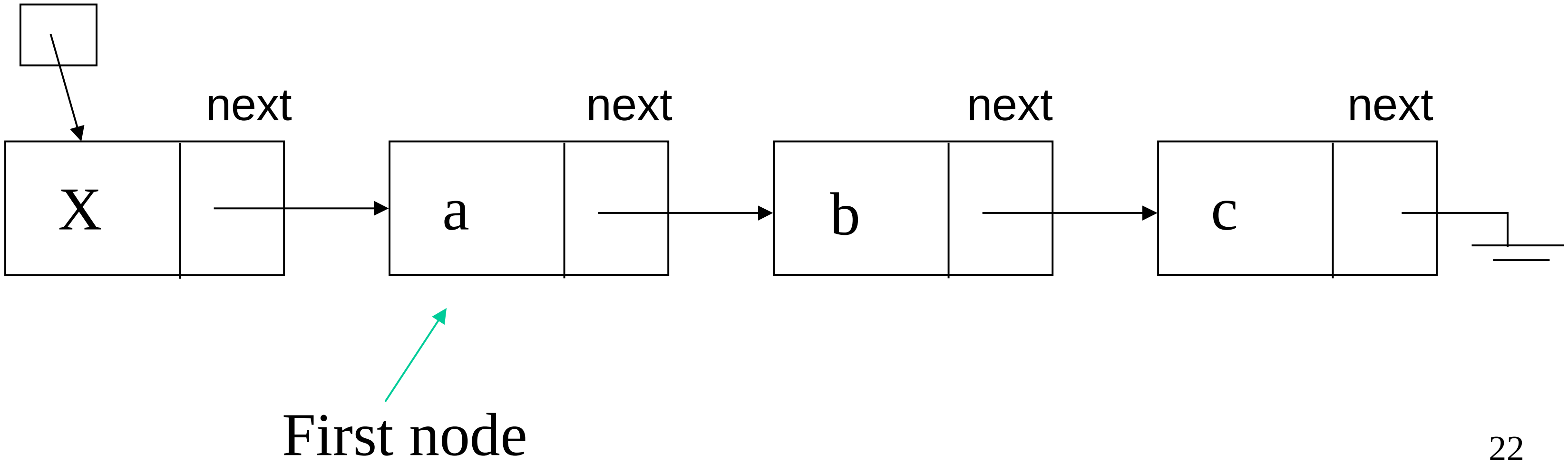
```
template <class T>
void List<T>::insert(const T& data, Node<T>* p) {

    if (p != NULL) { // case 1
      Node<T>* newNode = new Node<T>(data, p->next);
      p->next = newNode;
    }
    else { // case 2
      Node<T>* newNode = new Node<T>(data, head);
      head = newNode;
    }
}
```

Linked List Operations

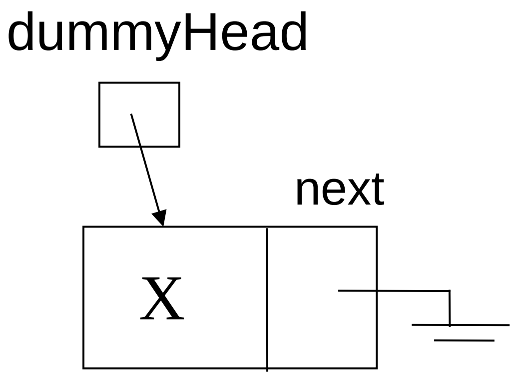
- To avoid this if-check at every insertion, we can add a **dummy head** to our list (also useful for deletion)
- This dummy head will be our zeroth node and its next pointer will point to the actual head (first node)

dummyHead (refer to this as `head' in code – next slides)



Linked List Operations

- An empty list will look like this (the contents of the node is irrelevant):



Linked List Operations

- Now, insertion code is simplified:

```
template <class T>
void List<T>::insert(const T& data, Node<T>* p) {

  // now p should not be NULL. To insert to the
  // first position, it should point to dummy head

  Node<T>* newNode = new Node<T>(data, p->next);
  p->next = newNode;

}
```

Linked List Operations

- We must make some changes to support the dummy head version:

```
template <class T>
class List {
private:
  Node<T> *dummyHead;
public:
  List() {
    dummyHead = new Node<T>(T(), NULL);
  }
};
```

Linked List Operations

```
template <class T>
class List {
private:
  Node<T> *dummyHead;
public:
  Node<T>* zeroth() {
      return dummyHead;
  }
  Node<T>* first() {
      return dummyHead->next;
  }
  const Node<T>* first() const {
      return dummyHead->next;
  }
  bool isEmpty() const {first() == NULL;}
};
```

“Note that if we don't have a constant first() function, we cannot make isEmpty const as well”

Searching for an Element

- To find an element, we must loop through all elements until we find the element or we reach the end:

```
template <class T>
class List {
private:
  Node<T> *dummyHead;
public:
  ...
  Node<T>* find(const T& data);
  ...
};
```

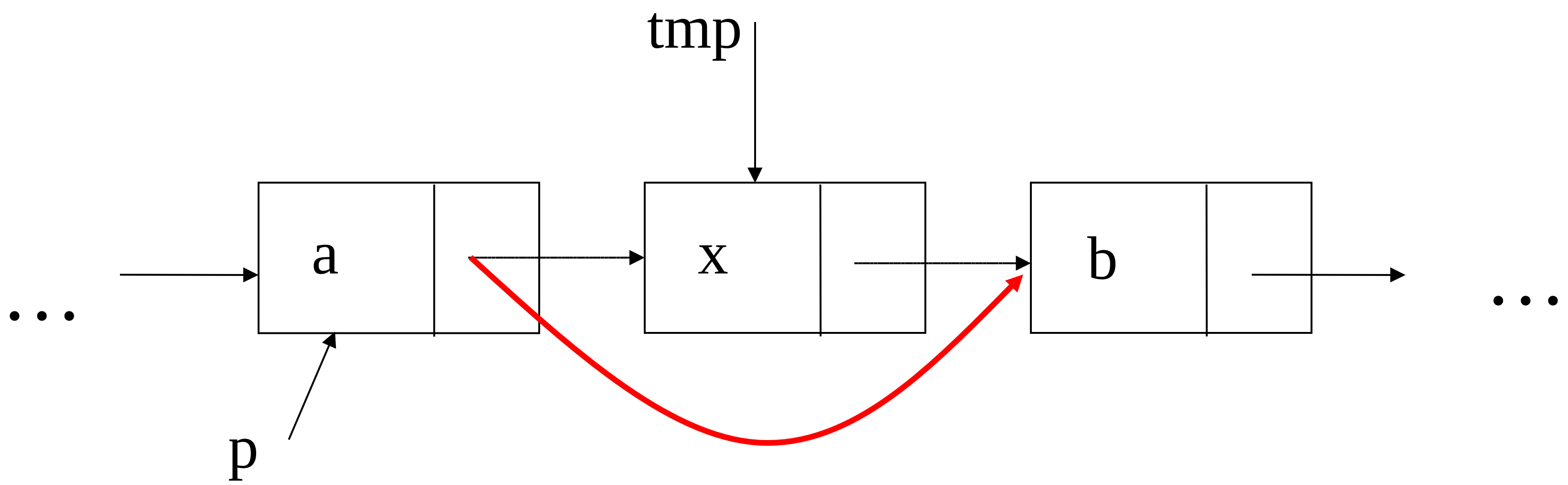
Searching for an Element

- To find an element, we must loop through all elements until we find the element or we reach the end:

```
template <class T>
Node<T>* List<T>::find(const T& data) {
  Node<T>* p = first();

  while (p) {
    if (p->element == data)
      return p;
    p = p->next;
  }
  return NULL;
}
```

Removing a node from a linked list



Removing an Element

- To remove a node containing an element, we must find the previous node of that node. So we need a **findPrevious** function

```
template <class T>
class List {
private:
  Node<T> *dummyHead;
public:
   ...
  Node<T>* findPrevious(const T& data);
   ...
};
```

Finding Previous Node

```
template <class T>
Node<T>* List<T>::findPrevious(const T& data)
{
  Node<T>* p = zeroth();

  while (p->next) {
    if (p->next->element == data)
      return p;
    p = p->next;
  }
  return NULL;
}
```

Removing an Element

- Now we can implement the remove function:

```
template <class T>
class List {
private:
  Node<T> *dummyHead;
public:
  ...
  void remove(const T& data);
  ...
};
```

Removing an Element

- Note that, because we have a dummy head, removal of an element is simplified as well

```
template <class T>
void List<T>::remove(const T& data) {
  Node<T>* p = findPrevious(data);

  if (p) {
    Node<T>* tmp = p->next;
    p->next = tmp->next;
    delete tmp;
  }
}
```

Printing All Elements (Traversal)

- List traversal is straightforward:

```
template <class T>
class List {
private:
  Node<T> *dummyHead;
public:
  ...
  void print() const;
  ...
};
```

Printing All Elements (Traversal)

- List traversal is straightforward:

"Again, the constant first() function allows print() to be const as well"

```
template <class T>
void List<T>::print() const
{
  const Node<T>* p = first();

  while(p) {
    std::cout << p->element << std::endl;
    p = p->next;
  }
}
```

Removing All Elements

- We can make the list empty by deleting all nodes (except the dummy head)

```
template <class T>
class List {
private:
  Node<T> *dummyHead;
public:
  ...
  void makeEmpty();
  ...
};
```

Removing All Elements

```
template <class T>
void List<T>::makeEmpty()
{
  while(!isEmpty()) {
    remove(first()->element());
  }
}
```

Destructor

- We must release allocated memory in the destructor

```
template <class T>
class List {
private:
  Node<T> *dummyHead;
public:
  ...
  ~List();
  ...
};
```

Destructor

```
template <class T>
List<T>::~List()
{
  makeEmpty();

  delete dummyHead;
}
```

Assignment Operator

- As the list has pointer members, we must provide an assignment operator

```
template <class T>
class List {
private:
  Node<T> *dummyHead;
public:
  ...
  List& operator=(const List& rhs);
  ...
};
```

Assignment Operator

```
template <class T>
List<T>& List<T>::operator=(const List& rhs)
{
  if (this != &rhs) {
    makeEmpty();
    const Node<T>* r = rhs.first();
    Node<T>* p = zeroth();

    while (r) {
       insert(r->element, p);
       r = r->next;
       p = p->next;
    }
  }
  return *this;
}
```

Uses the const version

Copy Constructor

- Finally, we must implement the copy constructor

```
template <class T>
class List {
private:
  Node<T> *dummyHead;
public:
  ...
  List(const List& rhs);
  ...
};
```

Copy Constructor

```
template <class T>
List<T>::List(const List& rhs)
{
  dummyHead = new Node<T>(T(), NULL);

  *this = rhs; // use operator=
}
```

Testing the Linked List Class

- Let's check if our implementation works by implementing a test driver file

```
int main() {
    List<int> list;
    list.insert(0, list.zeroth());
    Node<int>* p = list.first();

    for (int i = 1; i <= 10; ++i)
    {
        list.insert(i, p);
        p = p->next;
    }

    std::cout << "printing original list" << std::endl;
    list.print();
```

Testing the Linked List Class

```
for (int i = 0; i <= 10; ++i)
{
    if (i % 2 == 0)
        list.remove(i);
}

std::cout << "printing odd number list" << std::endl;
list.print();
```

Testing the Linked List Class

```
List<int> list2 = list;
cout << "printing copy constructed list" << endl;
list2.print();

List<int> list3;
list3 = list;
cout << "printing assigned list" << endl;
list3.print();

list.makeEmpty();
cout << "printing emptied list" << endl;
list.print();
```

Testing the Linked List Class

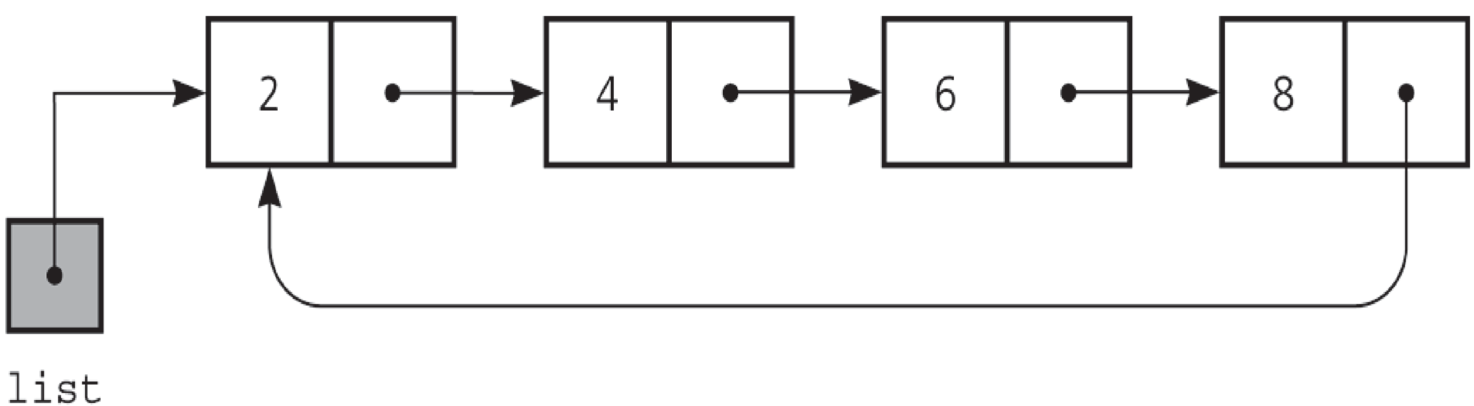
```
for (int i = 0; i <= 10; ++i) {
  if (i % 2 == 0) {
    if (list2.find(i) == NULL)
      cout << "could not find element " << i << endl;
  }
  else {
    if (list2.find(i) != NULL)
      cout << "found element " << i << endl;
  }
}
```

Variations of Linked Lists

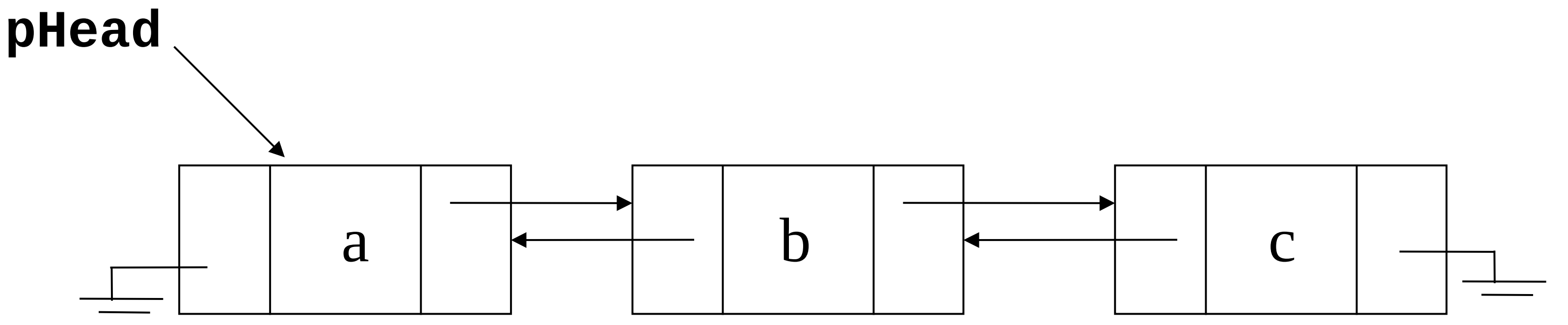
- The linked list that we studied so far is called **singly linked list**
- Other types of linked lists exist, namely:
  - Circular linked linked list
  - Doubly linked list
  - Circular doubly linked list
- Each type of linked list may be suitable for a different kind of application
- They may also use a dummy head for simplifying insertions and deletions

Circular Linked Lists

- Last node references the first node
- Every node has a successor
- No node in a circular linked list contains *NULL*
- **E.g.** a turn-based game may use a circular linked list to switch between players



Doubly Linked Lists



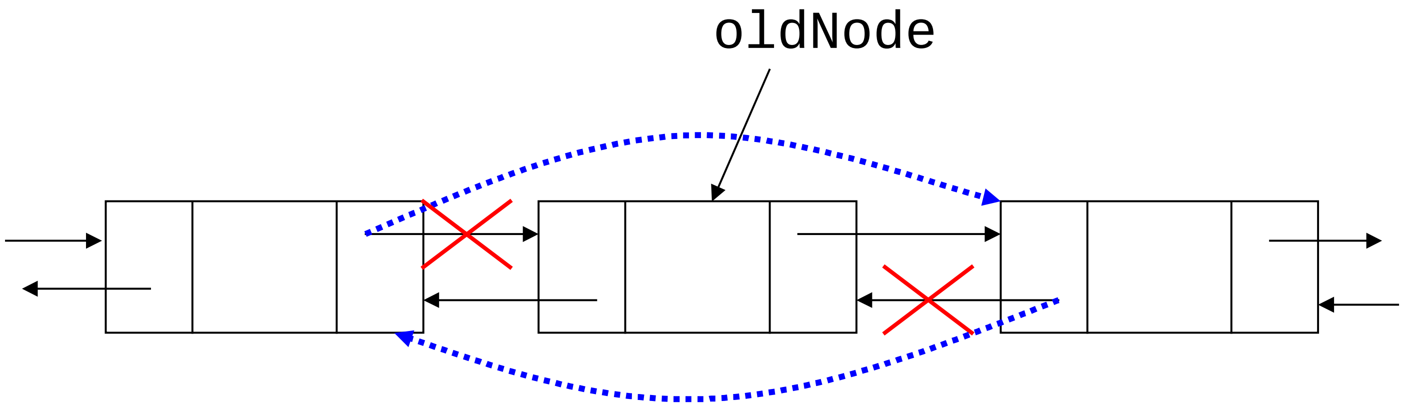
Advantages:

- Convenient to traverse the list backwards.
- **E.g.** printing the contents of the list in backward order

Disadvantage:

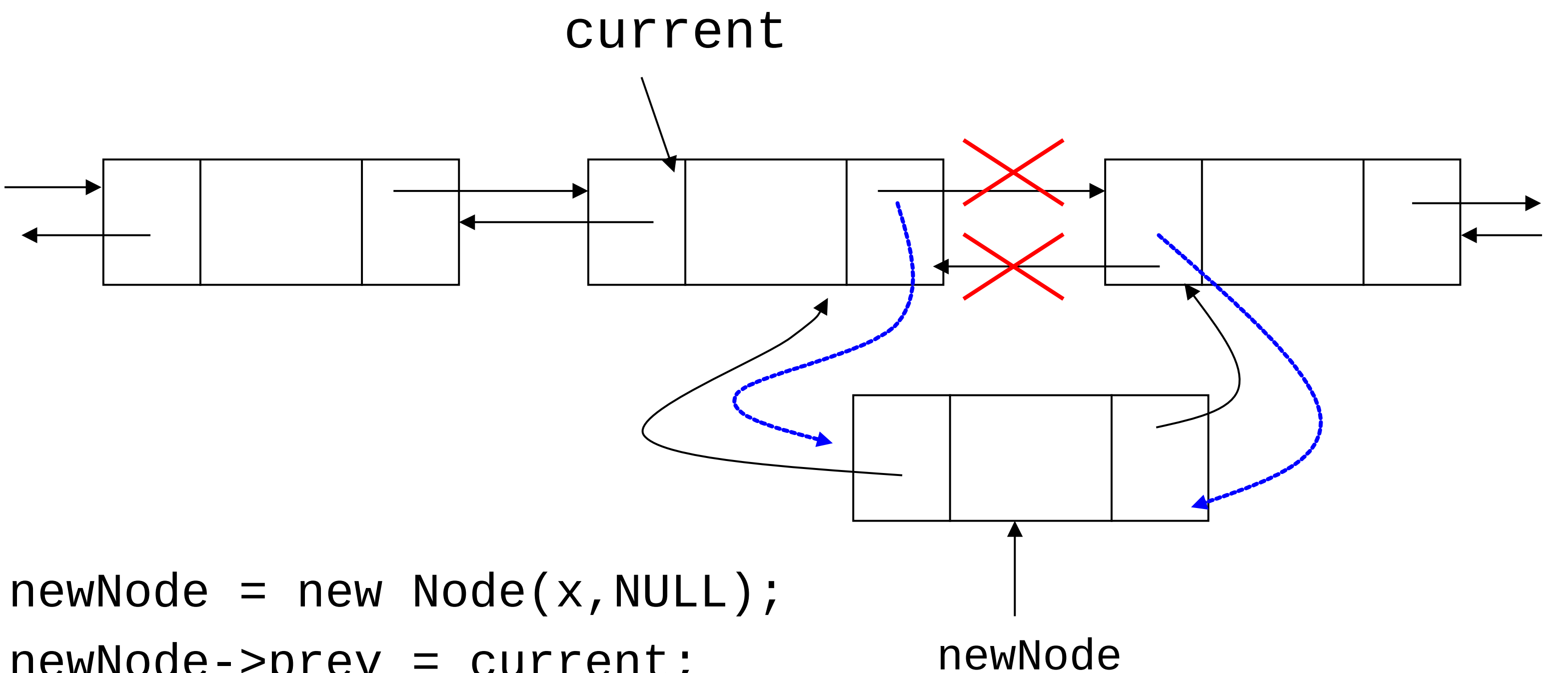
- Increase in space requirements due to storing two pointers instead of one

Deletion




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

Insertion

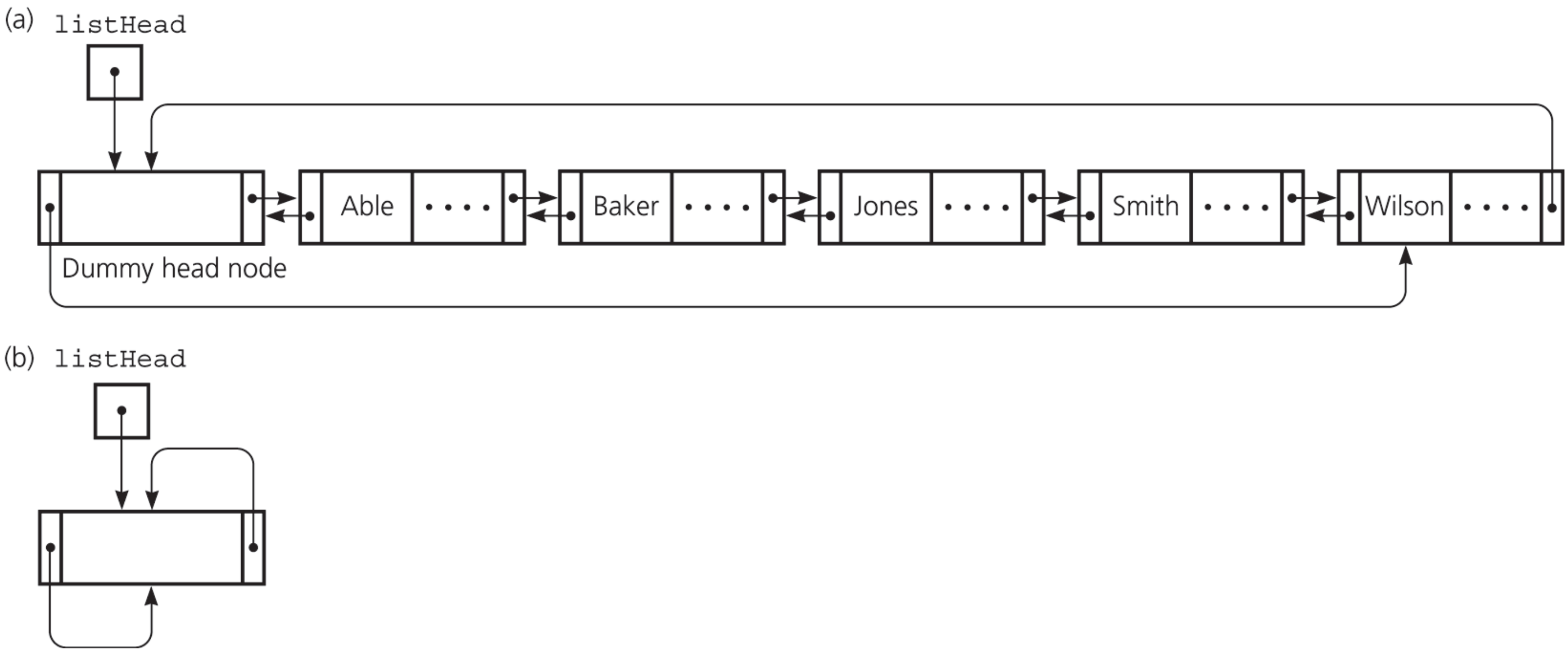



```
newNode = new Node(x,NULL);
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
```

Circular Doubly Linked Lists

- Circular doubly linked list
  - `prev` pointer of the dummy head node points to the last node
  - `next` reference of the last node points to the dummy head node
  - No special cases for insertions and deletions

Circular Doubly Linked Lists



(a) A circular doubly linked list with a dummy head node

(b) An empty list with a dummy head node