List (P.1)
Array List and Singly Linked List

Data Structures and Algorithms

Dept. Computer Science
Faculty of Computer Science and Engineering
Ho Chi Minh University of Technology, VNU-HCM

Linear list concepts
List ADT
Array implementation
How to store?
Implementation in C++
Singly linked list
Conceptual idea
Implementation in C++
Operations
Comparison of implementations

Overview

1. **Linear list concepts**
   - List ADT

2. **Array implementation**
   - How to store?
   - Implementation in C++

3. **Singly linked list**
   - Conceptual idea
   - Implementation in C++
   - Operations

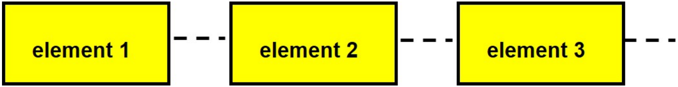
4. **Comparison of implementations**

Linear list concepts

Linear list concepts

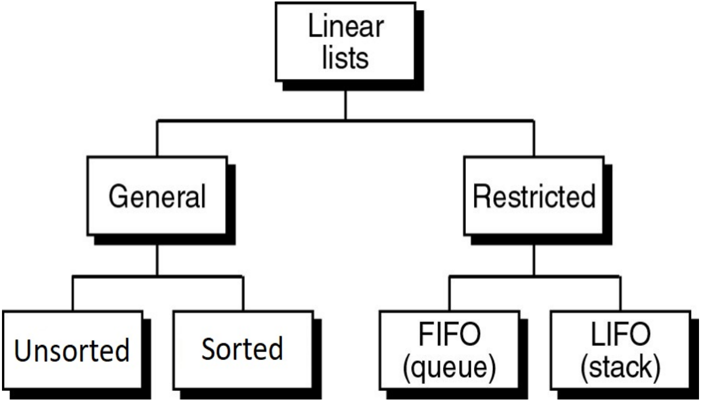
Definition

A linear list is a finite, ordered sequence of data items known as elements. "Ordered" in this definition means that each element has a position in the list.



Linear list concepts
List ADT
Array implementation
How to store?
Implementation in C++
Singly linked list
Conceptual idea
Implementation in C++
Operations
Comparison of implementations

Linear list concepts



Linear list concepts

General list:

- No restrictions on which operation can be used on the list.
- No restrictions on where data can be inserted/deleted.
- Unsorted list: data are not arranged in particular order.
- Sorted list: data are arranged according to a key.

Linear list concepts

Restricted list:

- Only some operations can be used on the list.
- Data can be inserted/deleted only at the ends of the list.
- Queue: FIFO (First-In-First-Out).
- Stack: LIFO (Last-In-First-Out).

List ADT

Definition

A list of elements of type T is a finite, ordered sequence of elements of T.

Basic concepts:

- A list is empty when it contains no elements.
- The number of elements currently stored is called the length (size) of the list.
- The beginning of the list is called the head, the end of the list is called the tail.

List ADT

Basic operations:

- Construct a list, leaving it empty.
- Insert an element.
- Remove an element.
- Search an element.
- Retrieve an element.
- Traverse the list, performing a given operation on each element.

Linear list concepts
List ADT
Array implementation
How to store?
Implementation in C++
Singly linked list
Conceptual idea
Implementation in C++
Operations
Comparison of implementations

List ADT

Extended operations:

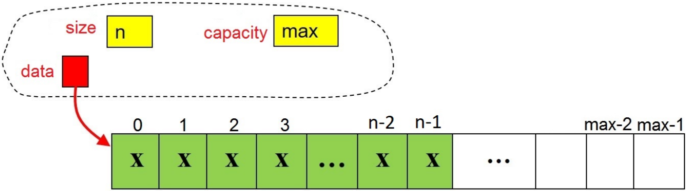
- Determine whether the list is empty or not.
- Determine whether the list is full or not.
- Find the size of the list.
- Clear the list to make it empty.
- Replace an element with another element.
- Merge two ordered list.
- Append an unordered list to another.

List ADT: Implementation in C++

```cpp
template <class T>
class IList
{
public:
    virtual void add(T e) = 0;
    virtual void add(int index, T e) = 0;
    virtual T removeAt(int index) = 0;
    virtual bool removeItem(T item) = 0;
    virtual bool empty() = 0;
    virtual int size() = 0;
    virtual void clear() = 0;
    virtual T get(int index) = 0;
    virtual void set(int index, T e) = 0;
    virtual int indexOf(T item) = 0;
    virtual bool contains(T item) = 0;
    virtual string toString() = 0;
};
```

Array implementation

Dynamically Allocated Array



```
List // Contiguous Implementation of List
  // number of used elements (mandatory)
  count <integer>

  // (Dynamically Allocated Array)
  data <Array List of <DataType> >

  capacity <integer>
End List
```

Array List: Implementation in C++ with Integer

```cpp
class IntArrayList : public IList<int> {
protected:
    int *data;
    int capacity;
    int count;
public:
    IntArrayList();
    virtual ~IntArrayList();
    virtual void add(int element);
    virtual void add(int index, int element);
    virtual int removeAt(int index);
    virtual bool removeItem(int item);
    virtual bool empty();
    virtual int size();
    virtual void clear();
    virtual int get(int index);
    virtual void set(int index, int element);
    virtual int indexOf(int item);
    virtual bool contains(int item);
    virtual string toString();
    virtual void dump();
};
```

Array List: Implementation in C++ with Integer

```cpp
class IntArrayList : public IList<int>
{
// ...

private:
    void checkIndex(int index);
    void ensureCapacity(int capacity);
};
```

Array List: Implementation in C++ with Integer

```cpp
IntArrayList::IntArrayList()
{
    this->capacity = 10;
    this->data = new int[this->capacity];
    this->count = 0;
}

IntArrayList::~IntArrayList()
{
    delete[] this->data;
}
```

Array List: Implementation in C++ with Integer

```cpp
void IntArrayList::add(int element)
{
    this->ensureCapacity(this->count + 1);

    this->data[this->count] = element;
    this->count++;
}

void IntArrayList::add(int index, int element)
{
    this->checkIndex(index);
    this->ensureCapacity(this->count + 1);

    int moveCount = this->count - index;
    if (moveCount > 0)
        memmove(this->data + index + 1,
                this->data + index,
                moveCount * sizeof(int));

    this->data[index] = element;
    this->count++;
}
```

Array List: Implementation in C++ with Integer

```cpp
int IntArrayList::removeAt(int index) {
    this->checkIndex(index);
    int elementToRemove = this->data[index];

    int moveCount = this->count - index - 1;
    if (moveCount > 0)
        memmove(this->data + index,
                this->data + index + 1,
                sizeof(int) * moveCount);

    this->count--;
    return elementToRemove;
}

bool IntArrayList::removeItem(int element) {
    for (int index = 0; index < this->count; index++)
    {
        if (this->data[index] == element) {
            this->removeAt(index);
            return true;
        }
    }
    return false;
}
```

Array List: Implementation in C++ with Integer

```cpp
bool IntArrayList::empty()
{
    return this->count == 0;
}

int IntArrayList::size()
{
    return this->count;
}

void IntArrayList::clear()
{
    delete[] this->data;

    this->capacity = 10;
    this->data = new int[this->capacity];
    this->count = 0;
}
```

Array List: Implementation in C++ with Integer

```cpp
int IntArrayList::get(int index) {
    this->checkIndex(index);

    return this->data[index];
}

void IntArrayList::set(int index, int element) {
    this->checkIndex(index);

    this->data[index] = element;
}

int IntArrayList::indexOf(int element) {
    for (int index = 0; index < this->count; index++)
        if (this->data[index] == element)
            return index;
    return -1;
}

bool IntArrayList::contains(int element) {
    return this->indexOf(element) != -1;
}
```

Array List: Implementation in C++ with Integer

```
string IntArrayList::toString() {
    stringstream ss;
    ss << "[";
    for (int index = 0; index < count - 1; index++)
        ss << data[index] << ",␣";

    if (count > 0) ss << data[count - 1] << "]";
    else ss << "]";

    return ss.str();
}

void IntArrayList::dump() {
    string line(50, '=');
    cout << line << endl;
    cout << "Integer␣list's␣information:" << endl;
    cout << "-␣Capacity:␣" << this->capacity << endl;
    cout << "-␣Size:␣" << this->count << endl;
    cout << "-␣Data:␣" << this->toString() << endl;
    cout << line << endl;
}
```

Array List: Implementation in C++ with Integer

```
void IntArrayList::checkIndex(int index) {
    if (index < 0 || index >= this->count)
        throw std::out_of_range(
                    "Index␣is␣out␣of␣range");
}

void IntArrayList::ensureCapacity(int capacity) {
    if (capacity > this->capacity) {
        int newCapacity = this->capacity * 3 / 2;
        int *newData = new int[newCapacity];
        memmove(newData, this->data,
                        this->count * sizeof(int));
        this->capacity = newCapacity;
        delete[] this->data;

        this->data = newData;
    }
}
```

Contiguous Implementation of List

In processing a contiguous list with n elements:

- Insert and Remove operate in time approximately proportional to n (require physical shifting).
- Clear, Empty, Full, Size, Replace, and Retrieve in constant time.

Singly linked list

Linked List

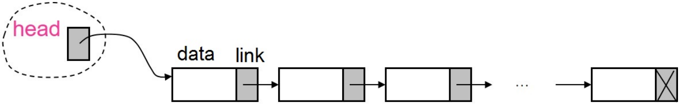


Figure: Singly Linked List

Structure

```
list // Linked Implementation of List
    head <pointer>
    tail <pointer>  // (optional)
    count <integer> // number of elements (optional)
end list
```

Nodes

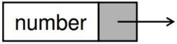
Definition

The elements in a linked list are called nodes.

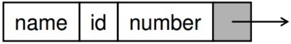
A node in a linked list is a structure that has at least two fields:

- the data,
- the address of the next node.

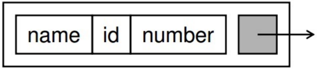
A node with one data field



A node with three data fields



A node with one structured data field



Nodes

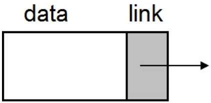
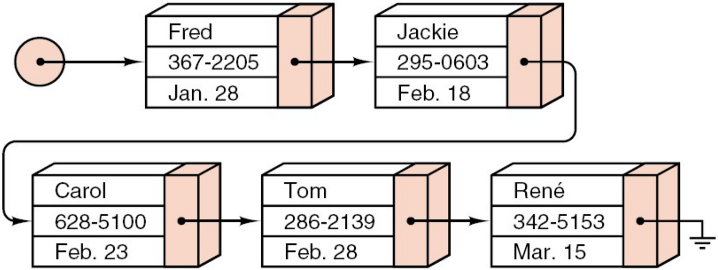


Figure: Linked list node structure

```
node
    data <dataType>
    link <pointer>
end node
```

```
// General dataType:
dataType
    key <keyType>
    field1 <...>
    field2 <...>
    ...
    fieldn <...>
end dataType
```

Example


Fred
367-2205
Jan. 28
Jackie
295-0603
Feb. 18
Carol
628-5100
Feb. 23
Tom
286-2139
Feb. 28
René
342-5153
Mar. 15

Example

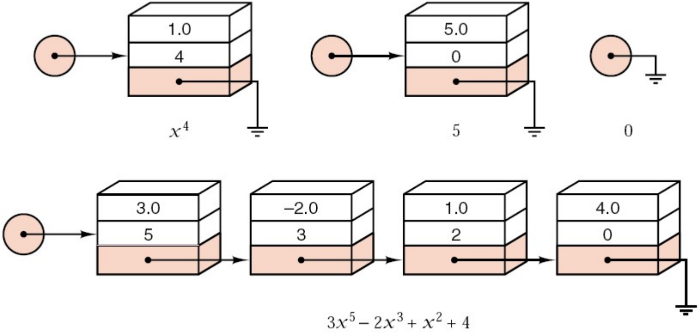


Figure: List representing polynomial

Implementation in C++

Example

```
node
    data <dataType>
    next <pointer>
end node
```

```
struct Node {
   int data;
   Node *linknext;
};
```

Implementation in C++ with `struct`

Example

```cpp
struct Node {
   int data;
   Node *next;
};
```

```cpp
struct Node {
   float data;
   Node *next;
};
```

In general, with `template`:

```cpp
template <class T>
struct Node {
   T data;
   Node<T> *next;
};
```

Node implementation in C++ with nested class

```cpp
class IntSLinkedList : public IList<int> {
public:
    class Node; // Forward declaration

protected:
    Node* head;
    Node* tail;
    int count;

public:
    IntSLinkedList() :
        head(NULL), tail(NULL), count(0) {};

public:
    class Node {
    protected:
        int data;
        Node *next;

    public:
        Node(int data = 0) {
            int data = 0;
            this->next = NULL;
        }
    }
};
```

Linked list operations

- Create an empty linked list
- Insert an item into a linked list
- Remove an item from a linked list
- Traverse a linked list
- Destroy a linked list
- ...

Insertion: Implementation in C++

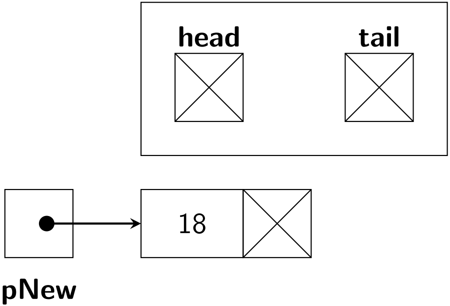
```cpp
class IntSLinkedList : public IList<int> {
// Declaration of attributes.
// Declaration of constructor, destructor.
// Declaration of nested classes.
public:
    virtual void add(int element);
    virtual void add(int index, int element);
};
```

Insert an item into a linked list

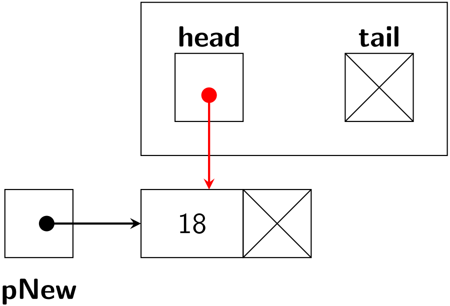
Cases with insertion:

1. Case 1: Empty list
2. Case 2: Non-empty list
   - Prepend an item into a list
   - Append an item into a list
   - Insert an item at a specific position in list

Insertion: Prepend to an empty list

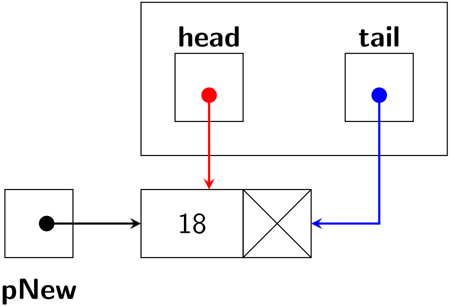


Insertion: Prepend to an empty list

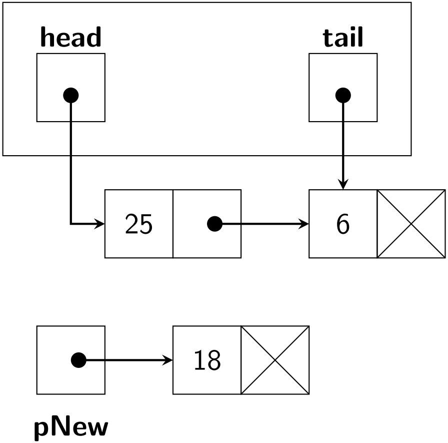


Linear list concepts
List ADT
Array implementation
How to store?
Implementation in C++
Singly linked list
Conceptual idea
Implementation in C++
Operations
Comparison of implementations

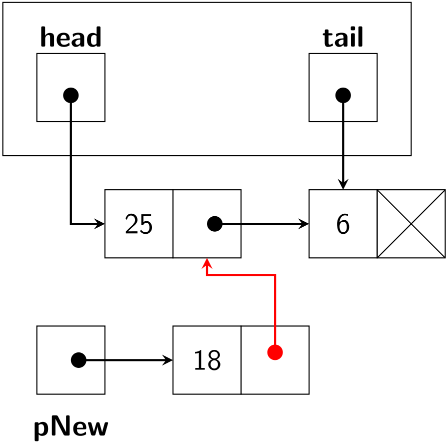
Insertion: Prepend to an empty list



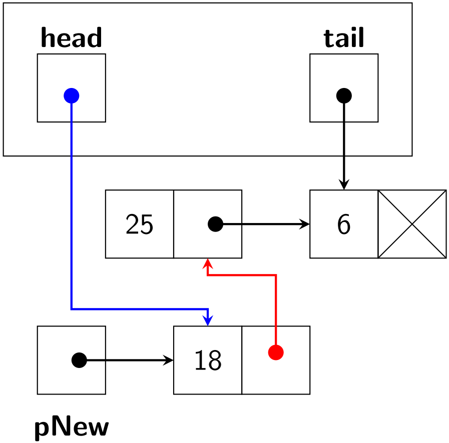
Insertion: Prepend to a non-empty list



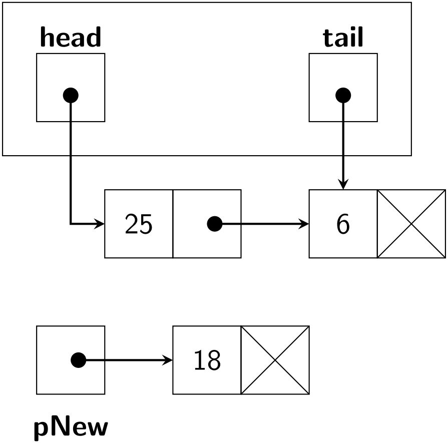
Insertion: Prepend to a non-empty list



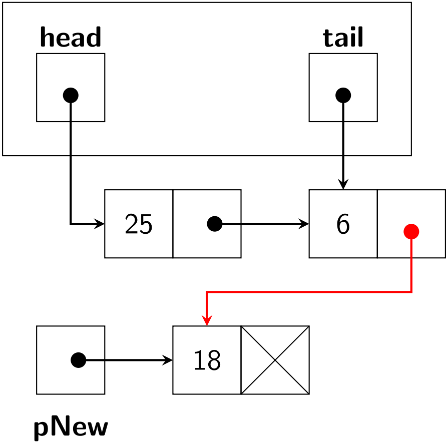
Insertion: Prepend to a non-empty list



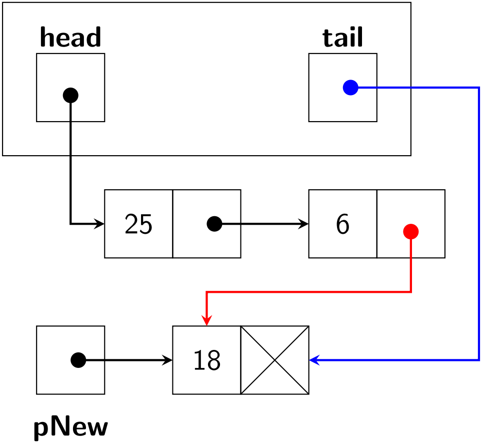
Insertion: Append to the list



Insertion: Append to the list

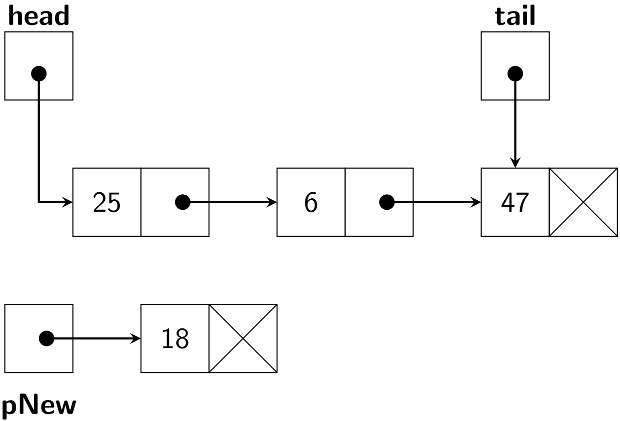


Insertion: Append to the list



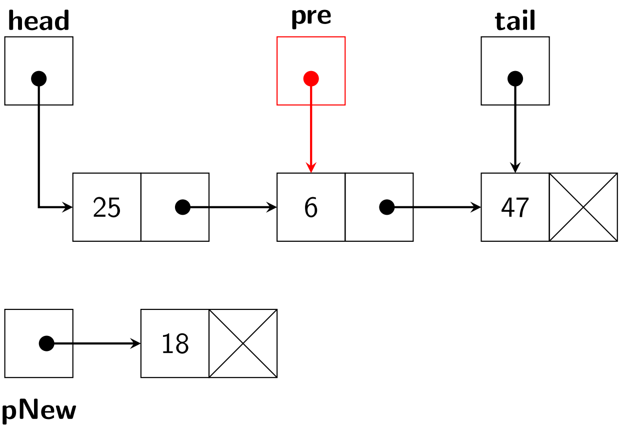
Insertion: At the index i

Insert **18** at index **2**.



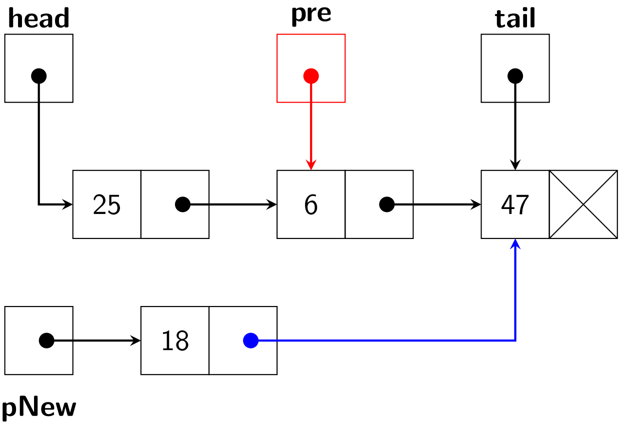
Insertion: At the index i

Insert **18** at index **2**.



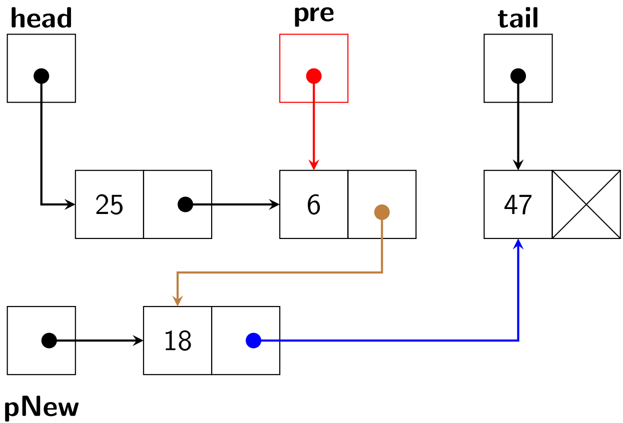
Insertion: At the index i

Insert **18** at index **2**.



Insertion: At the index i

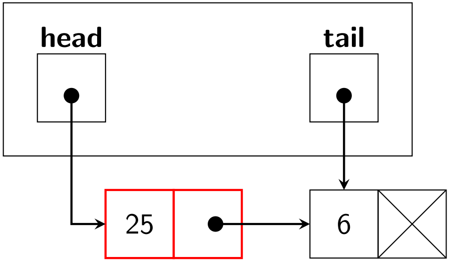
Insert **18** at index **2**.



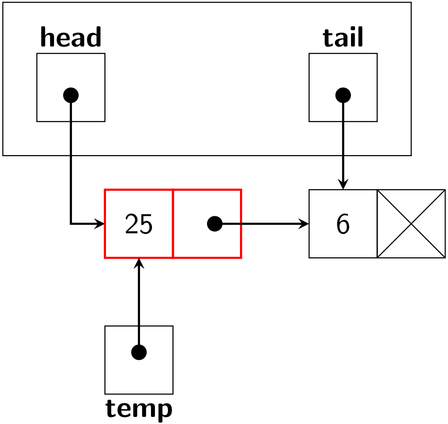
Removal: Implementation in C++

```cpp
class IntSLinkedList : public IList<int> {
// Declaration of attributes, constructor, destructor.
// Declaration of nested classes.
public:
    virtual int removeAt(int index);
    virtual bool removeItem(int item);
};
```

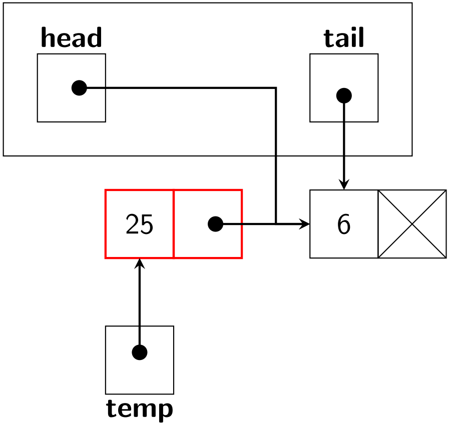
Removal: Delete the first element



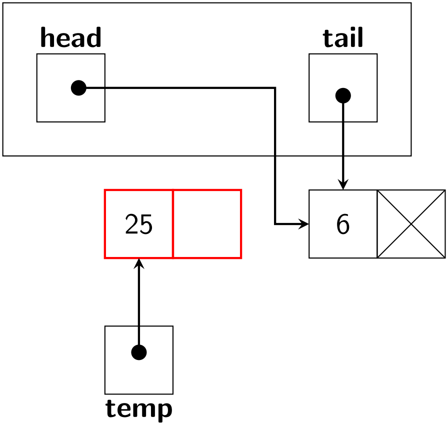
Removal: Delete the first element



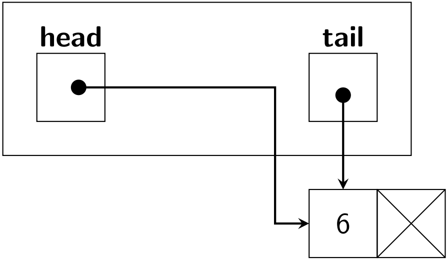
Removal: Delete the first element



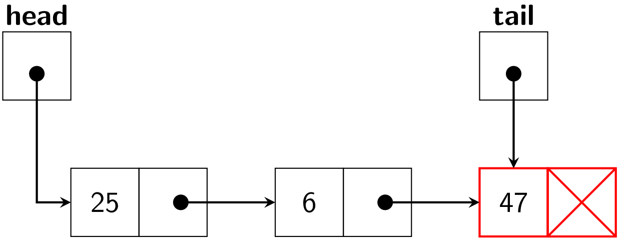
Removal: Delete the first element



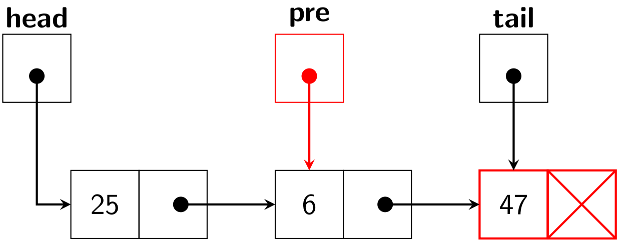
Removal: Delete the first element



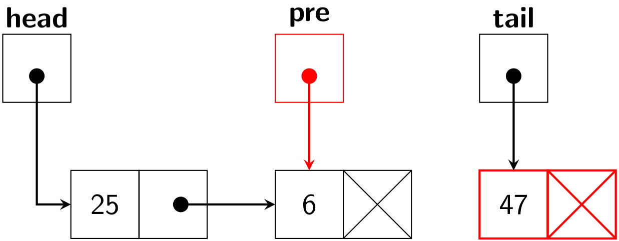
Removal: Delete the last element



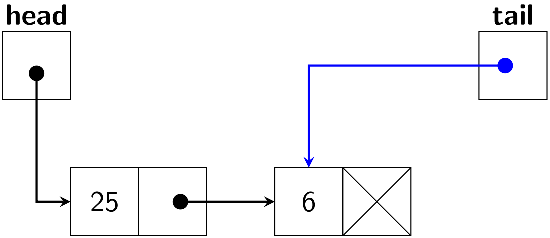
Removal: Delete the last element



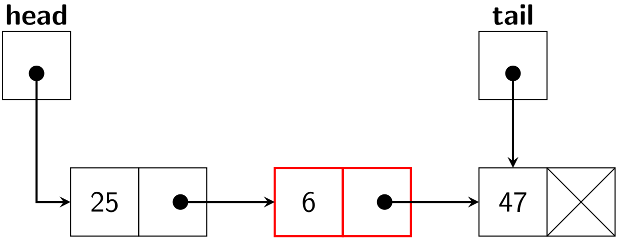
Removal: Delete the last element



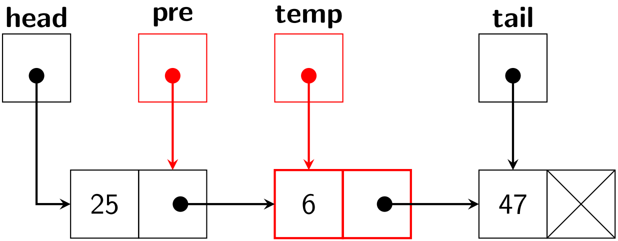
Removal: Delete the last element



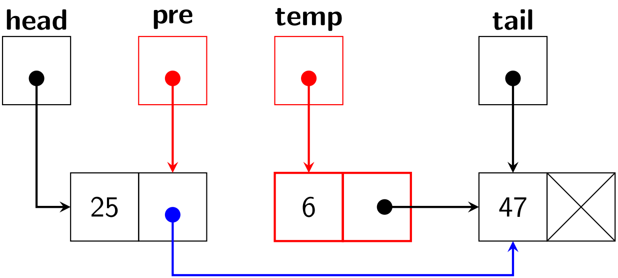
Removal: Delete at index i



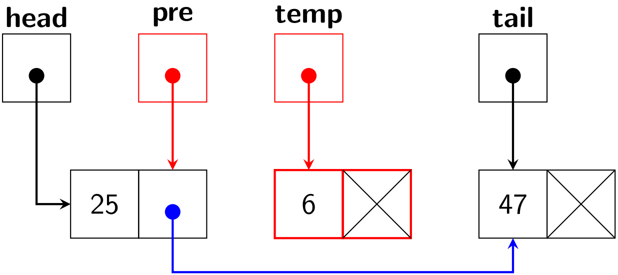
Removal: Delete at index i



Removal: Delete at index i

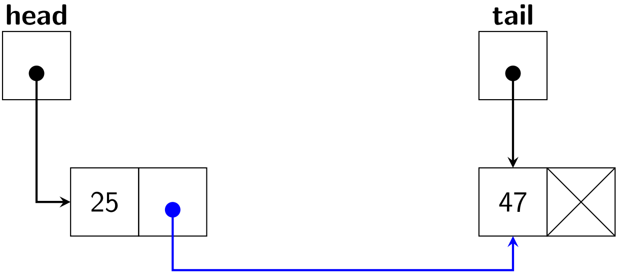


Removal: Delete at index i



Linear list concepts
List ADT
Array implementation
How to store?
Implementation in C++
Singly linked list
Conceptual idea
Implementation in C++
Operations
Comparison of implementations

Removal: Delete at index i

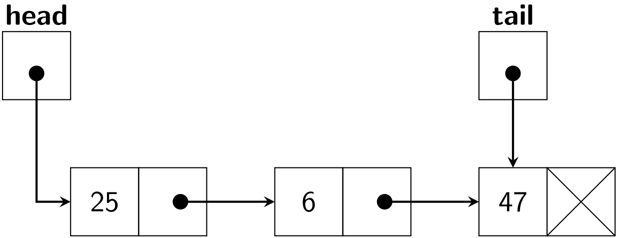


Search and other methods: Implementation in C++

```cpp
class IntSLinkedList : public IList<int> {
// Declaration of attributes.
// Declaration of constructor, destructor.
// Declaration of nested classes.
public:
    virtual int get(int index);
    virtual void set(int index, int element);
    virtual int indexOf(int item);
    virtual bool contains(int item);
};
```

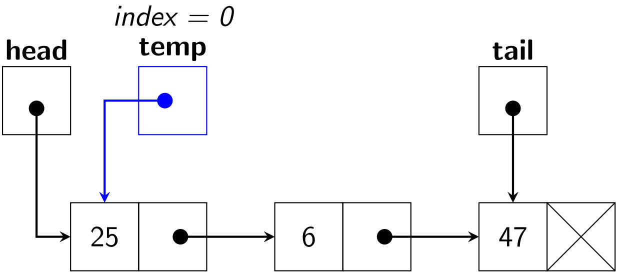
Searching

Search **element** with index **1**.



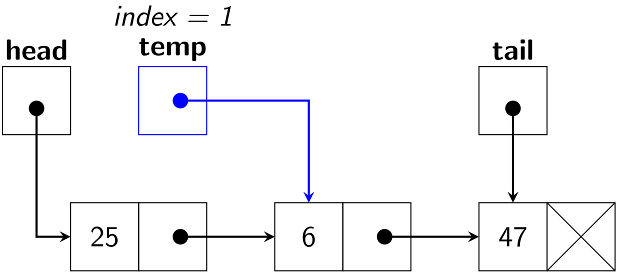
Searching

Search **element** with index **1**.



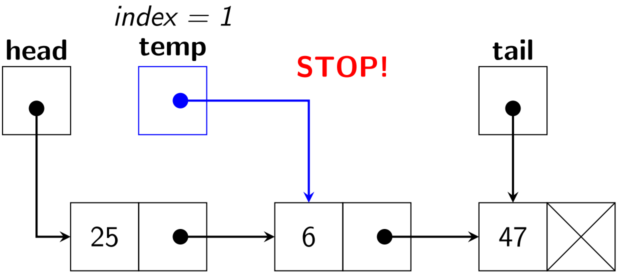
Searching

Search **element** with index **1**.



Searching

Search **element** with index **1**.



Comparison of implementations of list

Arrays: Pros and Cons

- **Pros**:
  - Access to an array element is fast since we can compute its location quickly.
- **Cons**:
  - If we want to insert or delete an element, we have to shift subsequent elements which slows our computation down.
  - We need a large enough block of memory to hold our array.

Linked Lists: Pros and Cons

- **Pros**:
  - Inserting and deleting data does not require us to move/shift subsequent data elements.
- **Cons**:
  - If we want to access a specific element, we need to traverse the list from the head of the list to find it which can take longer than an array access.

Comparison of implementations of list

- **Contiguous storage is generally preferable when**:
  - the entries are individually very small;
  - the size of the list is known when the program is written;
  - few insertions or deletions need to be made except at the end of the list; and
  - random access is important.
- **Linked storage proves superior when**:
  - the entries are large;
  - the size of the list is not known in advance; and
  - flexibility is needed in inserting, deleting, and rearranging the entries.

THANK YOU.