List

Lecturer:

Contact:

Faculty of Computer Science and Engineering
Hochiminh city University of Technology

Contents

Linear list concepts

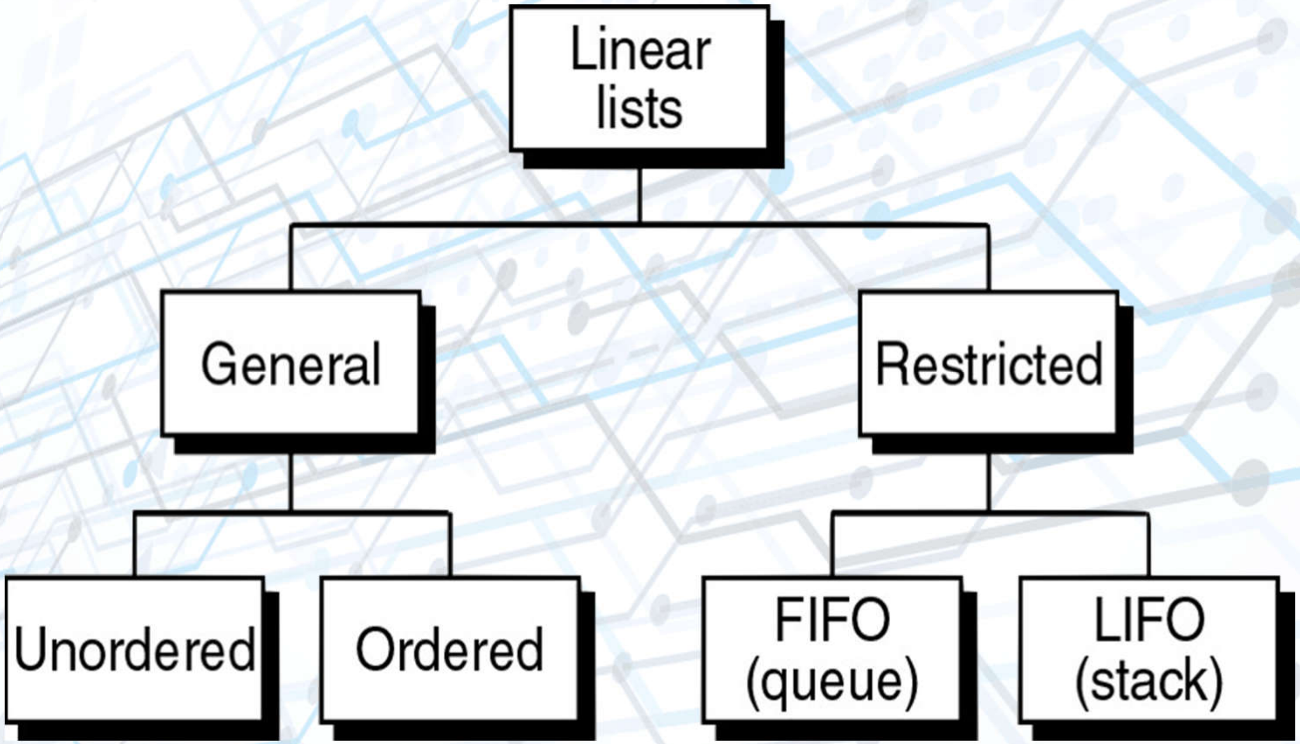
Linear list concepts

Definition
A linear list is a data structure in which each element has a unique successor.

Example

- Array
- Linked list

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.

- Unordered list (random list): Data are not in particular order.
- Ordered 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).

Definition
A list of elements of type T is a finite sequence of elements of type T.

Basic operations:

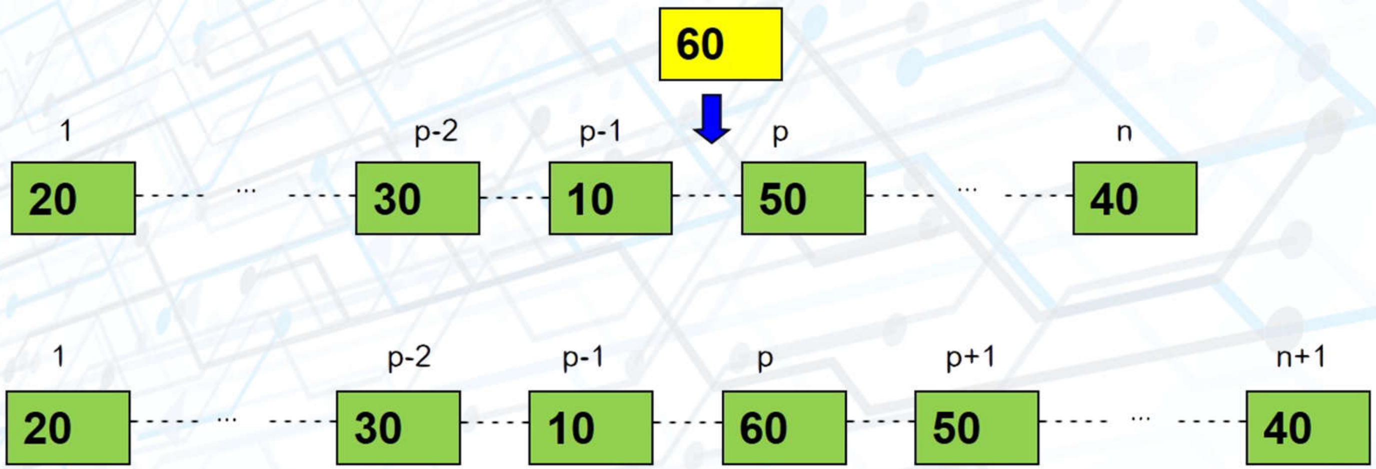
- Construct a list, leaving it empty.
- Insert an element.
- Remove an element.
- Search an element.
- Retrieve an element.
- Traverse the list, performing a given operation on each element.

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.

Insertion

- **Insert an element at a specified position p in the list**
  - Only with *General Unordered List*.



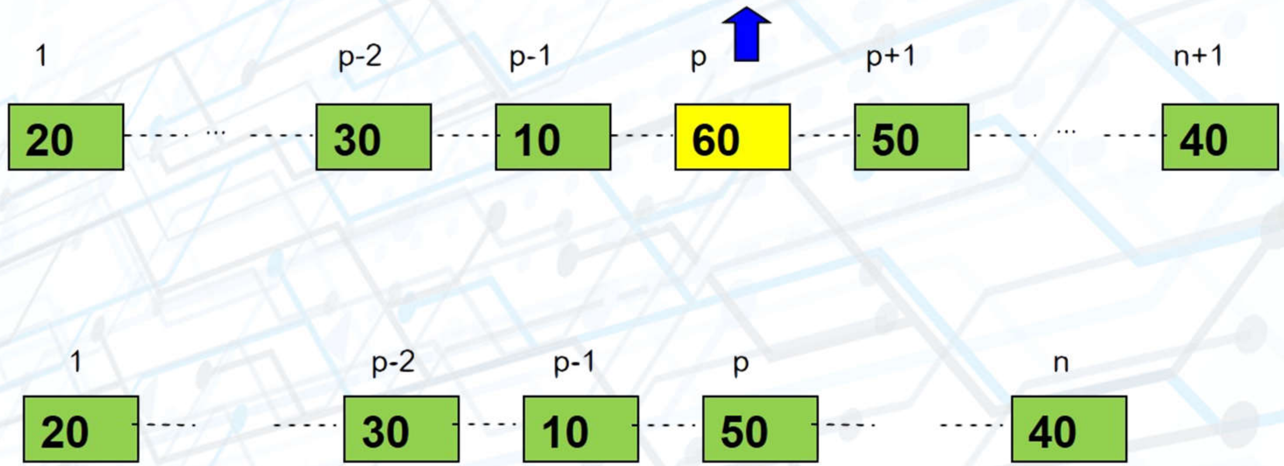
Any element formerly at position p and all later have their position numbers increased by 1.

Insertion

- **Insert an element with a given data**
  - With *General Unordered List*: can be made at any position in the list (at the beginning, in the middle, at the end).
  - With *General Ordered List*: data must be inserted so that the ordering of the list is maintained (searching appropriate position is needed).
  - With *Restricted List*: depend on it own definition (FIFO or LIFO).

Removal

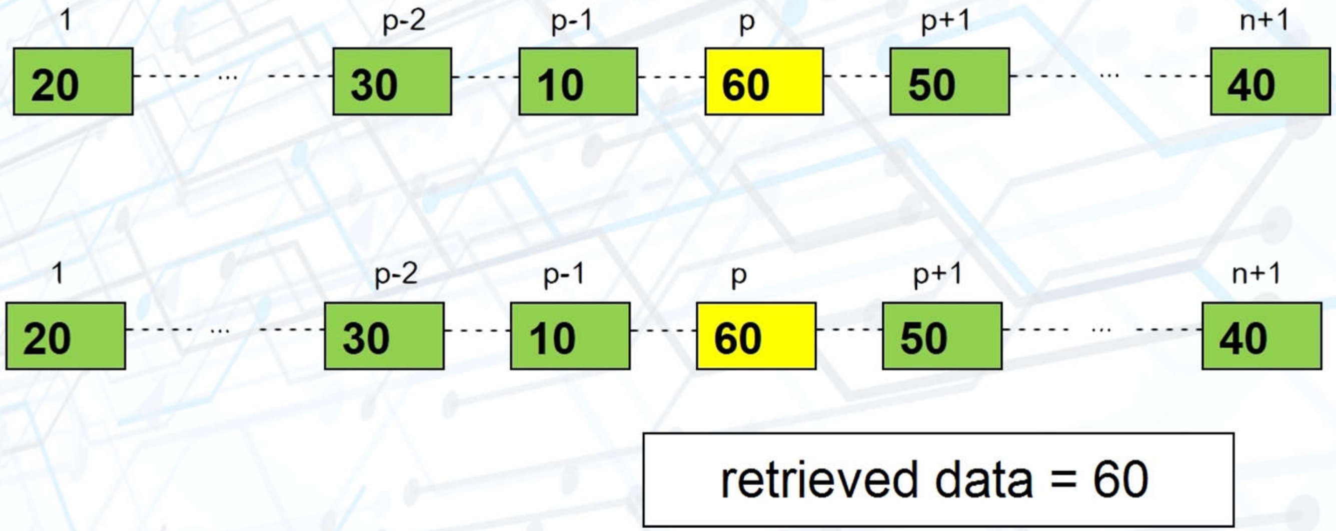
- **Remove an element at a specified position p in the list**
  - With *General Unordered List* and *General Ordered List*.



The element at position p is removed from the list, and all subsequent elements have their position numbers decreased by 1.

Retrieval

- **Retrieve an element at a specified position p in the list**
  - With *General Unordered List* and *General Ordered List*.



All elements remain unchanged.

Removal, Retrieval

- **Remove/ Retrieve an element with a given data**
  - With *General Unordered List* and *General Ordered List*: Searching is needed in order to locate the data being deleted/ retrieved.

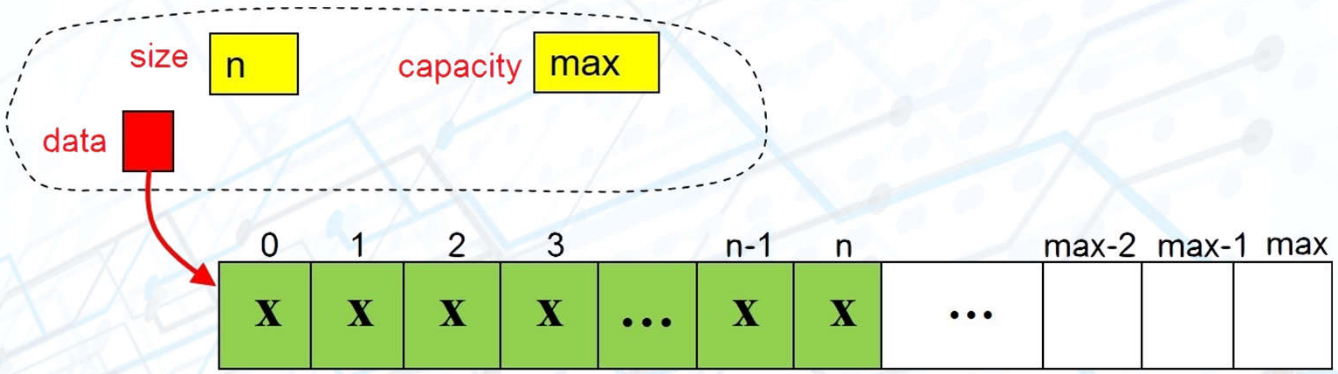
Success of Basic Operations

- **Insertion** is successful when the list is not full.

- **Removal, Retrieval** are successful when the list is not empty.

Array implementation

Dynamically Allocated Array




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

  // (Dynamically Allocated Array)
  data <dynamic array of <DataType> >

  capacity <integer>
End List
```

Dynamic Array: Implementation in C++

```
class DynamicArray {
private:
  int size;
  int capacity;
  int *storage;

public:     DynamicArray() {
    capacity = 10; size = 0;
    storage = new int[capacity];
  }
```

Dynamic Array: Implementation in C++

```
DynamicArray(int capacity) {
  this->capacity = capacity;
  size = 0;
  storage = new int[capacity];
}

~DynamicArray() {
  delete[] storage;
}
```

Dynamic Array: Implementation in C++

```
    void setCapacity(int);
    void ensureCapacity(int);
    void pack();
    void trim();

    void rangeCheck(int);
    void set(int, int);
    int get(int);
    void removeAt(int);
    void insertAt(int, int);

    void print();
};
```

Dynamic Array: Implementation in C++

```
void DynamicArray::setCapacity(int newCapacity) {
  int *newStorage = new int[newCapacity];
  memcpy(newStorage, storage,sizeof(int) * size);
  capacity = newCapacity;
  delete[] storage;
  storage = newStorage;
}
```

Dynamic Array: Implementation in C++

```
void DynamicArray::ensureCapacity(int minCapacity) {
  if (minCapacity > capacity) {
    int newCapacity = (capacity*3)/2 + 1;
    if (newCapacity < minCapacity)
        newCapacity = minCapacity;
    setCapacity(newCapacity);
  }
}
```

Dynamic Array: Implementation in C++

```
void DynamicArray::pack() {
  if (size <= capacity / 2) {
    int newCapacity = (size * 3) / 2 + 1;
    setCapacity(newCapacity);
  }
}

void DynamicArray::trim() {
  int newCapacity = size;
  setCapacity(newCapacity);
}
```

Dynamic Array: Implementation in C++

```
void DynamicArray::rangeCheck(int index) {
  if (index < 0 || index >= size)
      throw "Index out of bounds!";
}

void DynamicArray::set(int index, int value) {
  rangeCheck(index);
  storage[index] = value;
}

int DynamicArray::get(int index) {
  rangeCheck(index);
  return storage[index];
}
```

Dynamic Array: Implementation in C++

```
void DynamicArray::insertAt(int index, int value) {
    If (index < 0 || index > size)
        throw "Index out of bounds!";
    ensureCapacity(size + 1);

    int moveCount = size−index;
    if (moveCount != 0)
        memmove(storage + index + 1,
                storage + index,
                sizeof(int) * moveCount);
    storage[index]= value;
    size++;
}
```

Dynamic Array: Implementation in C++

```
void DynamicArray::removeAt(int index) {
  rangeCheck(index);
  int moveCount = size - index - 1;
  if (moveCount > 0)
     memmove(storage + index,
             storage + (index + 1),
             sizeof(int) * moveCount);
  size--;
  pack();
}
```

Dynamic Array: Using

```
void DynamicArray::print() {
    for (int i=0; i<this->size; i++) {
        cout << storage[i] << " ";
    }
}

int main() {
  cout << "Dynamic Array" << endl;
  DynamicArray* da = new DynamicArray(10);
  da->insertAt(0, 55);
  // ...
  da->print();
  return 0;
}
```

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

Definition

A linked list is an ordered collection of data in which each element contains the location of the next element.

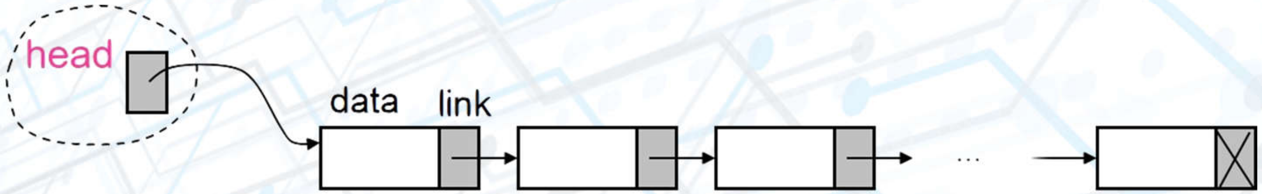


Figure 1: Singly Linked List

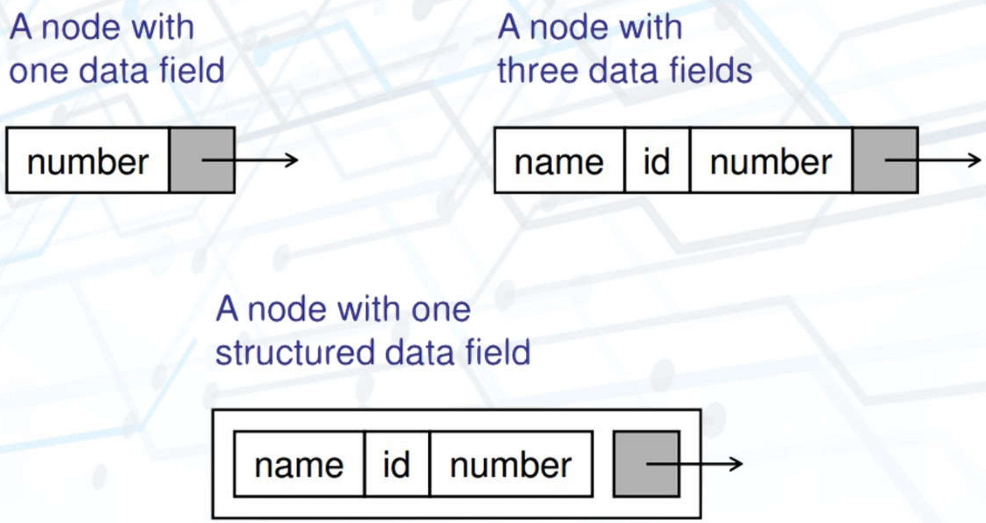
```
list // Linked Implementation of List
  head <pointer>
  count <integer> // number of elements (optional)
end list
```

Nodes

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.



Nodes

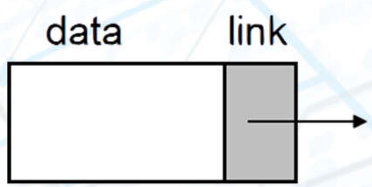
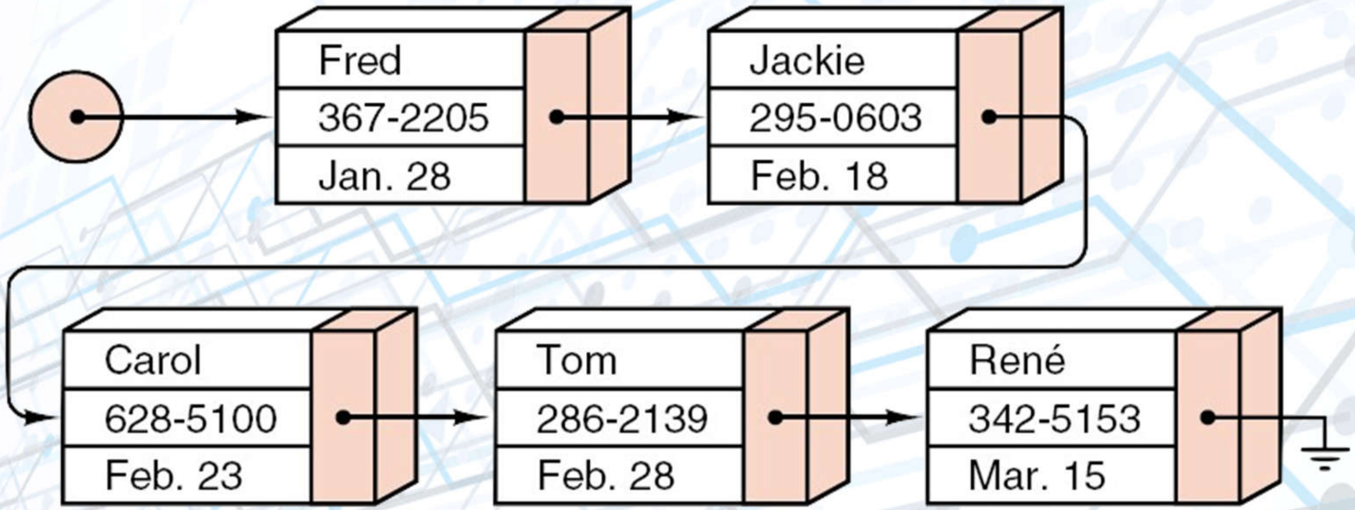


Figure 2: Linked list node structure

```
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

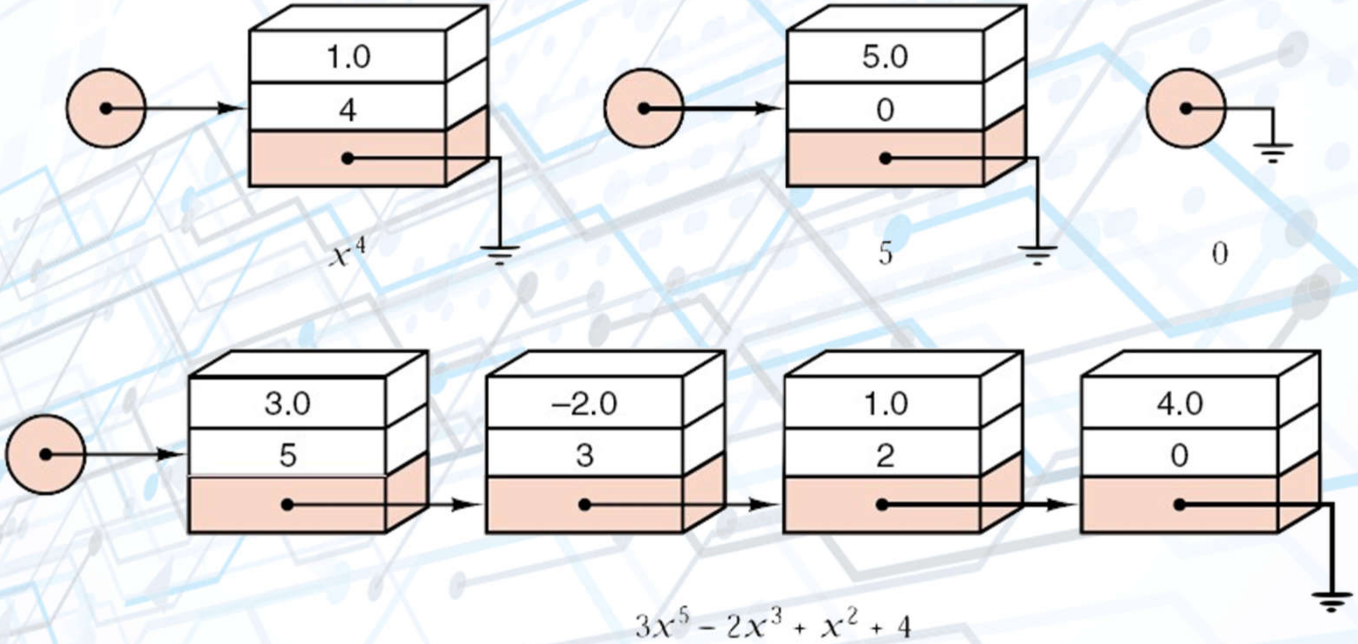


Figure 3: List representing polynomial

Implementation in C++

Example

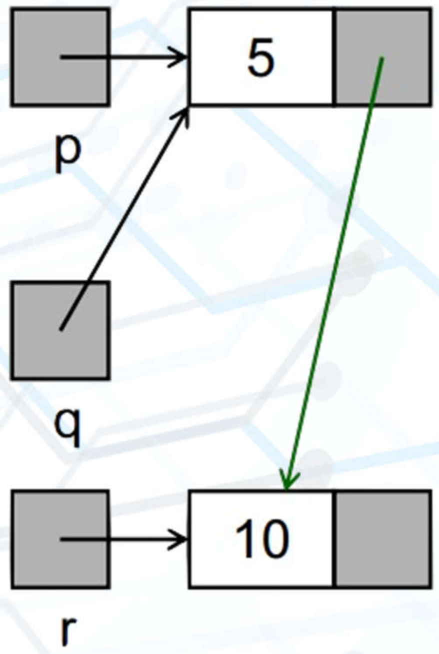
```
node
    data <dataType>
    link <pointer>
end node
```

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

Implementation in C + +

Example

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

struct Node{
    int data;
    Node *link;
};

int main(){
    Node*p=new Node();
    p->data=5;
    cout<<p->data<<endl;
    Node*q=p;
    cout<<q->data<<endl;
    Node *r=new Node();
    r->data=10;
    q->link=r;
    cout<<p->link->data<<endl;
}
```



Implementation in C++

Example

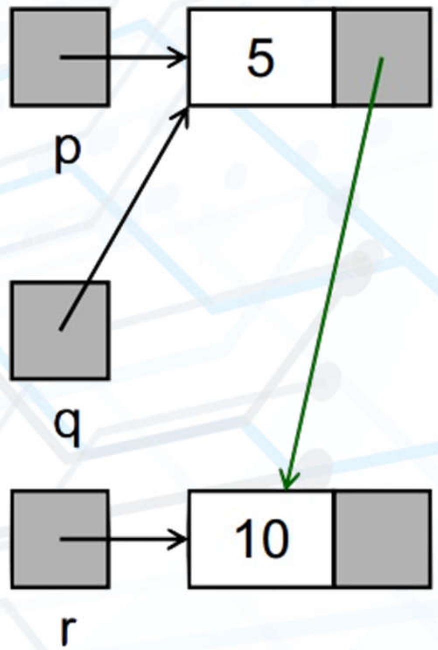
```
struct Node {
    int data;
    Node *link;
};
```

```
struct Node {
    float data;
    Node *link;
};
```

```
template <class ItemType>
struct Node {
    ItemType data;
    Node<ItemType> *link;
};
```

Example

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

template<class ItemType>
struct Node{
    ItemType data;
    Node <ItemType>*link;
};

int main(){
    Node<int>*p=new Node<int>();
    p->data=5;
    cout<<p->data<<endl;
    Node<int>*q=p;
    cout<<q->data<<endl;
    Node<int>*r=new Node<int>();
    r->data=10;
    q->link=r;
    cout<<p->link->data<<endl;
}
```



Node implementation in C++

```
template<class ItemType>
class Node{
    ItemType data;
    Node <ItemType>*link;
public:
    Node(){
        this->link=NULL;
    }
    Node(ItemType data){//constructor with initial value
        this->data=data;
        this->link=NULL;
    }
};
```

Linked list implementation in C++

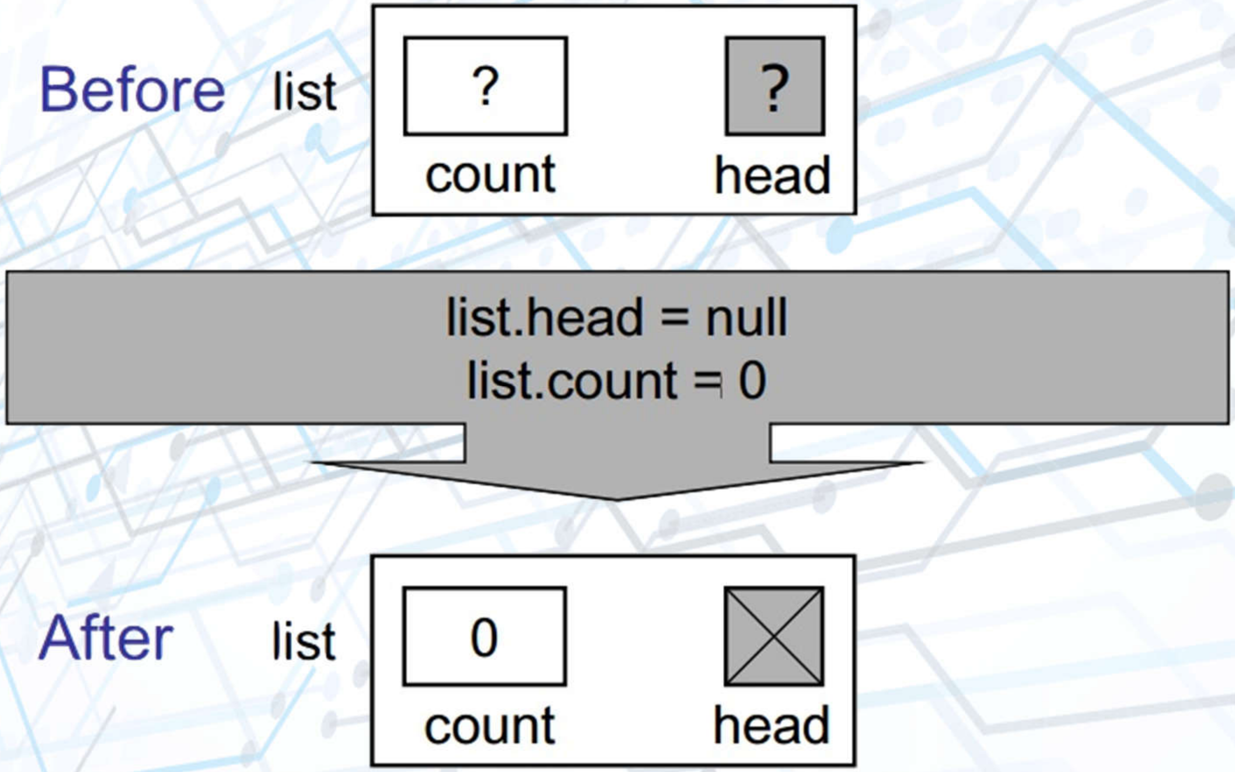
```
template<class List_ItemType>
class LinkedList{
    Node <List_ItemType>*head;
    int count;
public:
    LinkedList();
    ~LinkedList();
};
```

```
list
  head <pointer>
  count <integer>
end list
```

Linked list operations

- Create an empty linked list
- Insert a node into a linked list
- Delete a node from a linked list
- Traverse a linked list
- Destroy a linked list

Create an empty linked list



Create an empty linked list

Algorithm createList(ref list <metadata>)

Initializes metadata for a linked list

Pre: list is a metadata structure passed by reference

Post: metadata initialized

list.head = null

list.count= 0

return

End createList

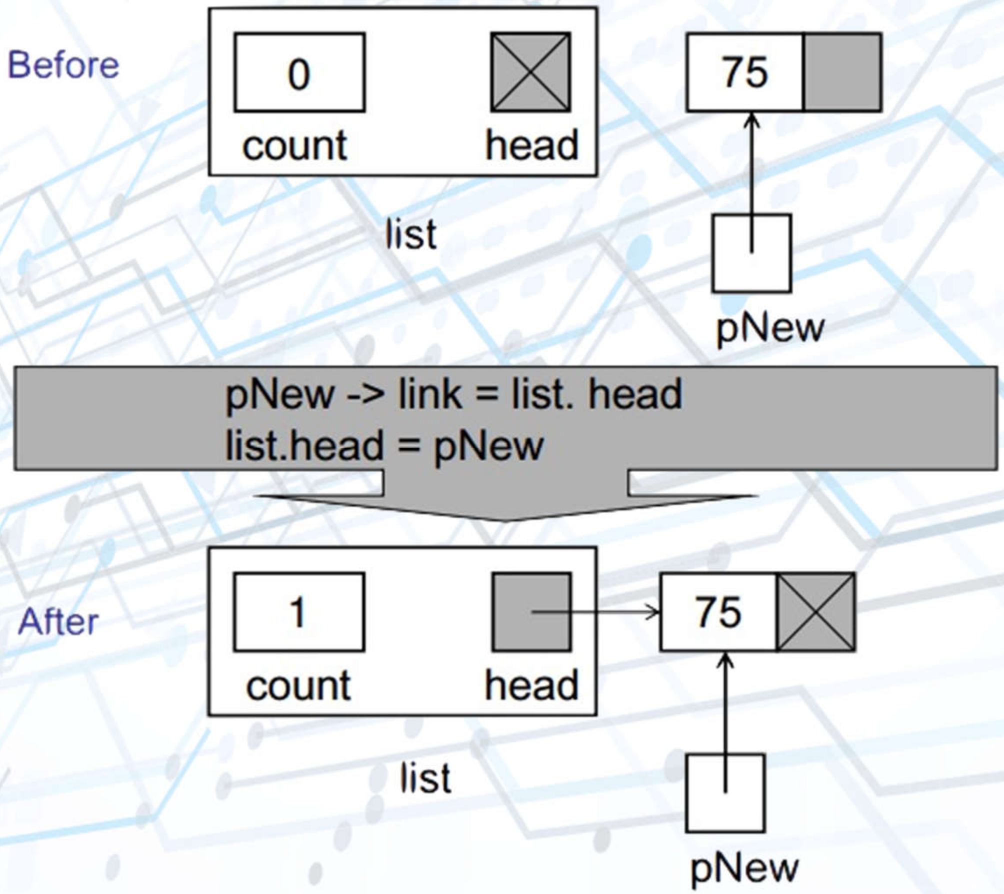
Create an empty linked list

```
template<class List_ItemType>
class LinkedList{
    Node<List_ItemType>*head;
    int count;
public:
    LinkedList();
    ~LinkedList();
};

template<class List_ItemType>
LinkedList<List_ItemType>::LinkedList(){
    this->head=NULL;
    this->count=0;
}
```

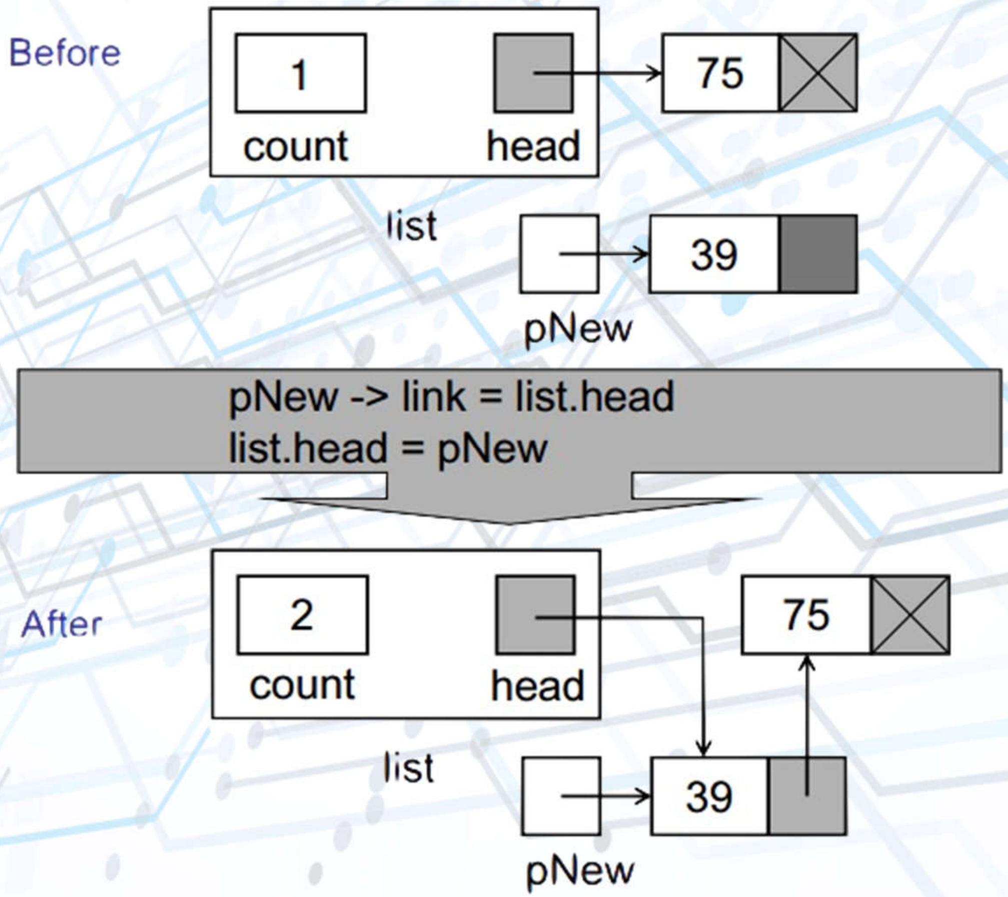
Insert a node into a linked list

1. Allocate memory for the new node and set up data.
2. Locate the pointer p in the list, which will point to the new node:
   - If the new node becomes the first element in the List: $\texttt{p is list.head}$.
   - Otherwise: $\texttt{p is pPre->link}$, where $\texttt{pPre}$ points to the predecessor of the new node.
3. Point the new node to its successor.
4. Point the pointer p to the new node.

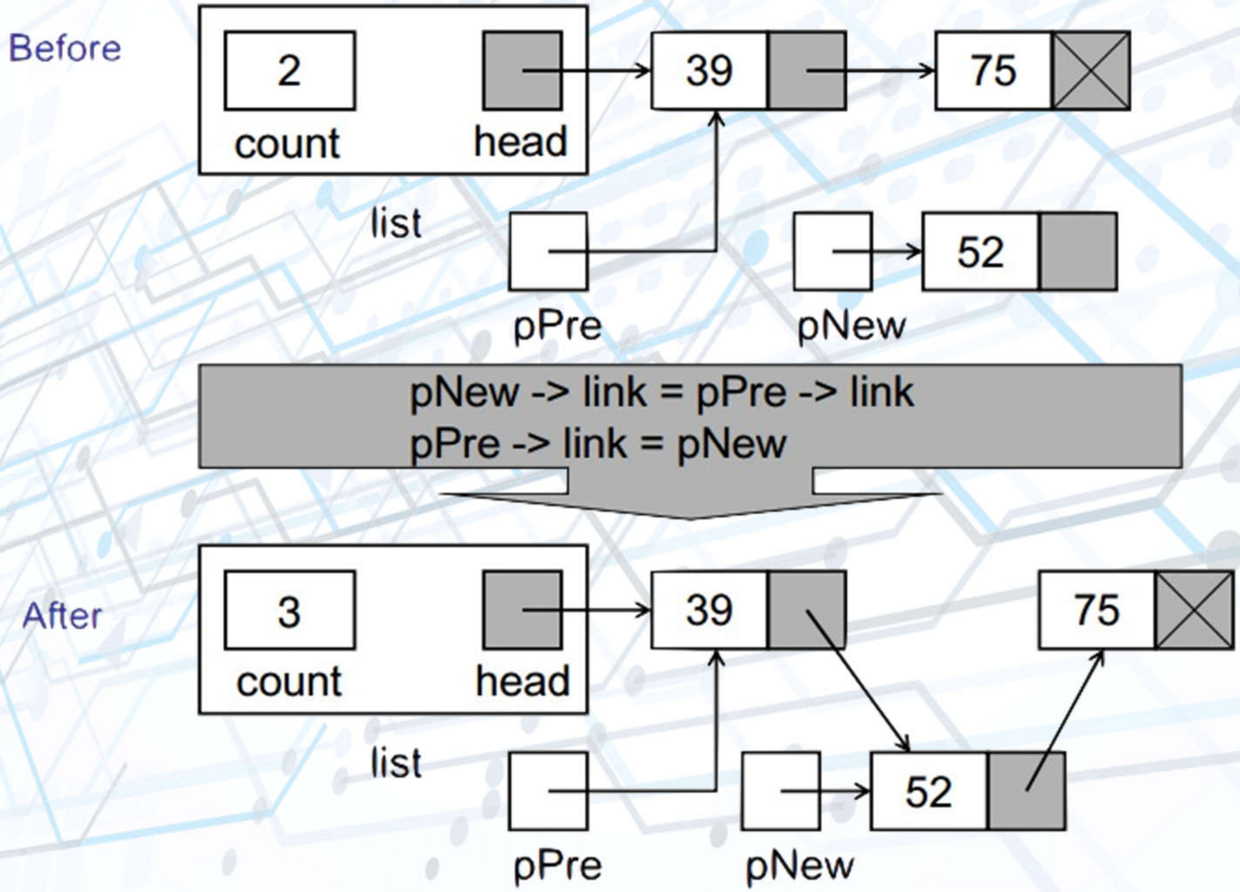
Insert into an empty linked list



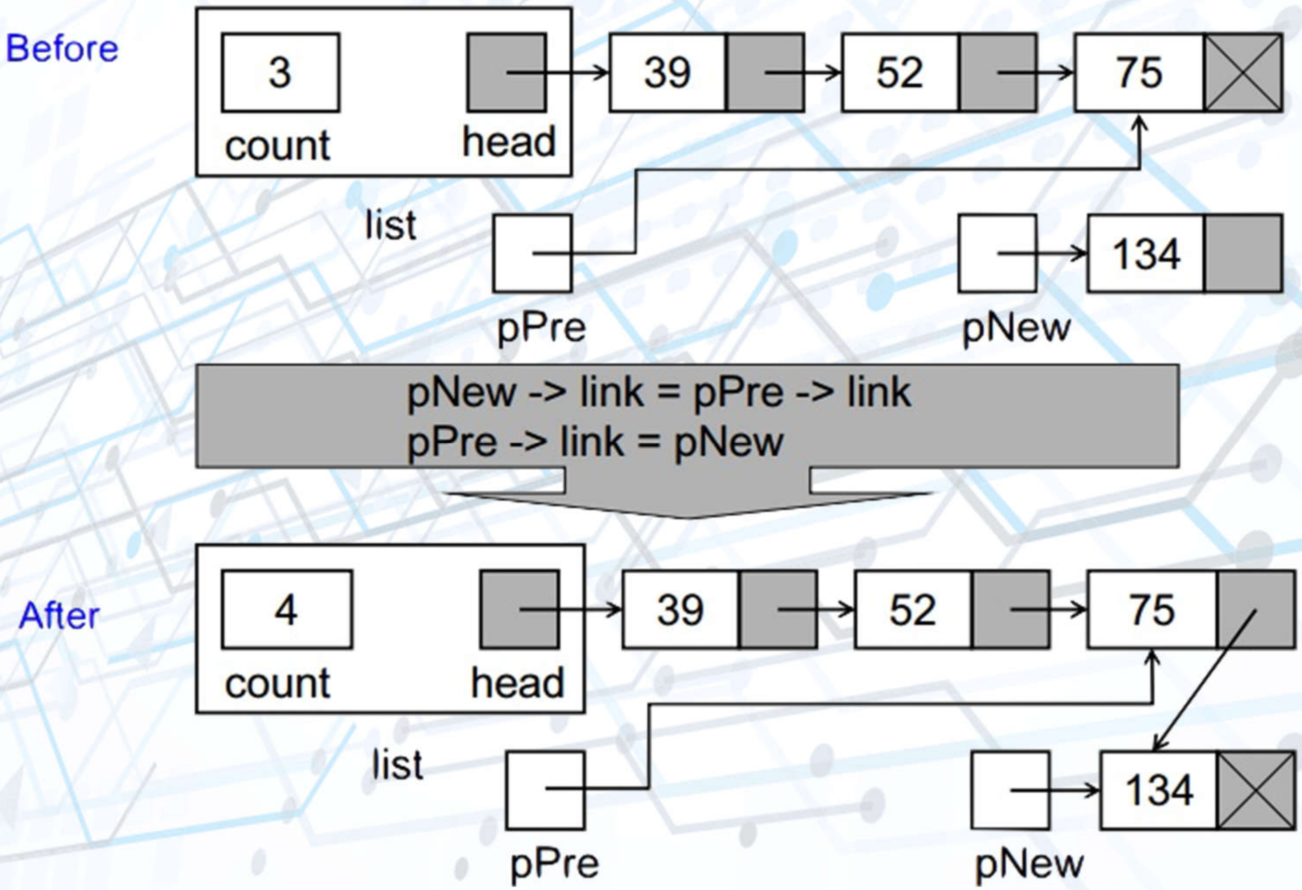
Insert at the beginning



Insert in the middle



Insert at the end



Insert a node into a linked list

- Insertion is successful when allocation memory for the new node is successful.
- There is no difference between insertion at the beginning of the list and insertion into an empty list.

```
pNew->link = list.head
list.head = pNew
```

- There is no difference between insertion in the middle and insertion at the end of the list.

```
pNew->link = pPre->link
pPre->link = pNew
```

Insert a node into a linked list

Algorithm insertNode(ref list <metadata>,
val pPre <node pointer>, val dataIn <dataType>)

Inserts data into a new node in the linked list.

Pre: list is metadata structure to a valid list pPre is pointer to data's logical predecessor dataIn contains data to be inserted

Post: data have been inserted in sequence

Return true if successful, false if memory overflow

Insert a node into a linked list

```
allocate(pNew)
if memory overflow then
    return false
end
pNew -> data = dataIn
if pPre = null then
    // Adding at the beginning or into empty list
    pNew -> link = list.head  list.head = pNew
else
    // Adding in the middle or at the end
    pNew -> link = pPre -> link  pPre -> link = pNew
end
list.count = list.count + 1  return true
End insertNode
```

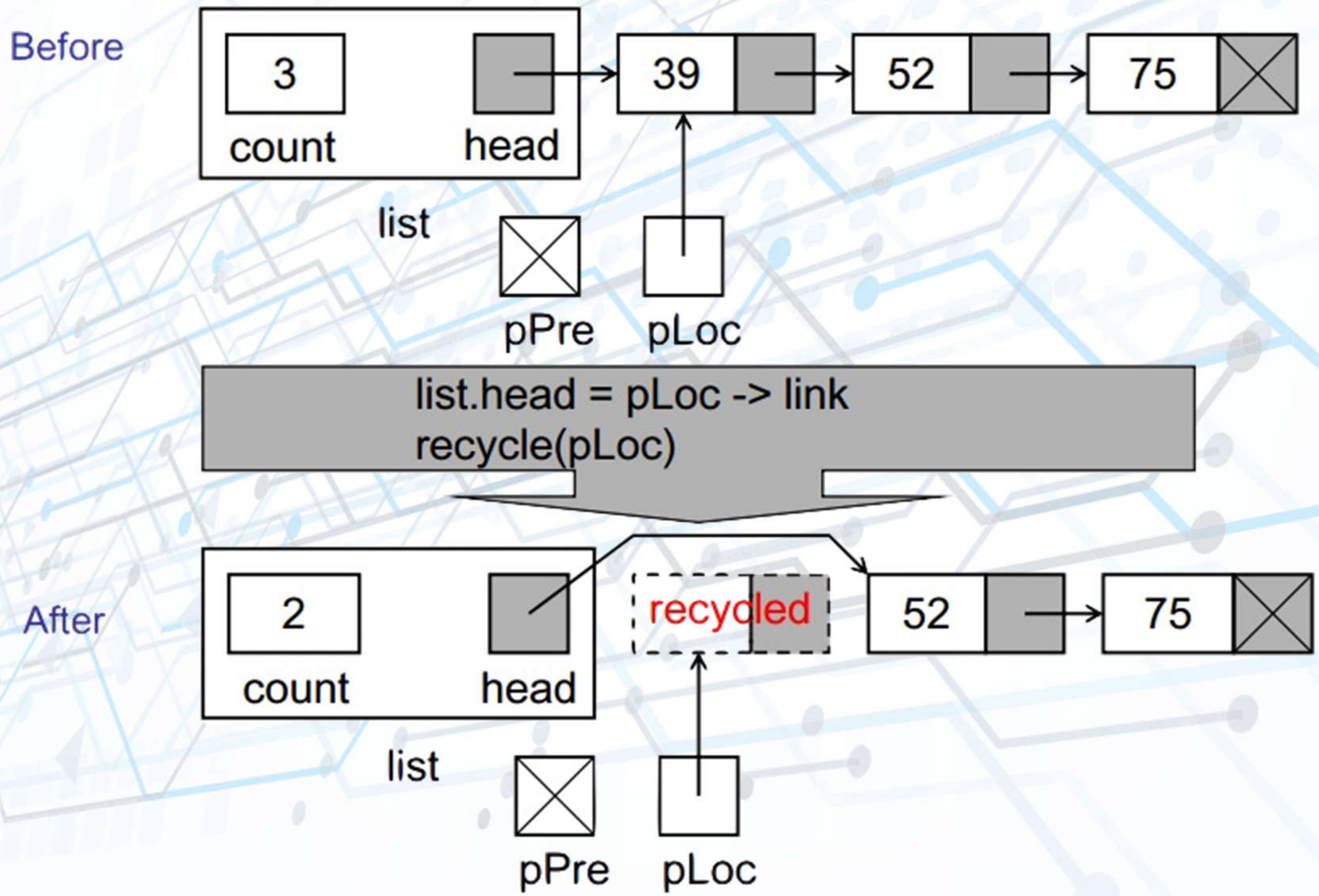
Insert a node into a linked list

```
template<class List_ItemType>
int LinkedList<List_ItemType>::InsertNode(Node <List_ItemType>*pPre,List_ItemType value){
    Node <List_ItemType>*pNew=new Node<List_ItemType>();
    if(pNew==NULL)
    return0;
    pNew->data=value;
    if(pPre==NULL){
        pNew->link=this->head;
        this->head=pNew;
    }
    else{
        pNew->link=pPre->link;
        pPre->link=pNew;
    }
    this->count++;
    return 1;
}
```

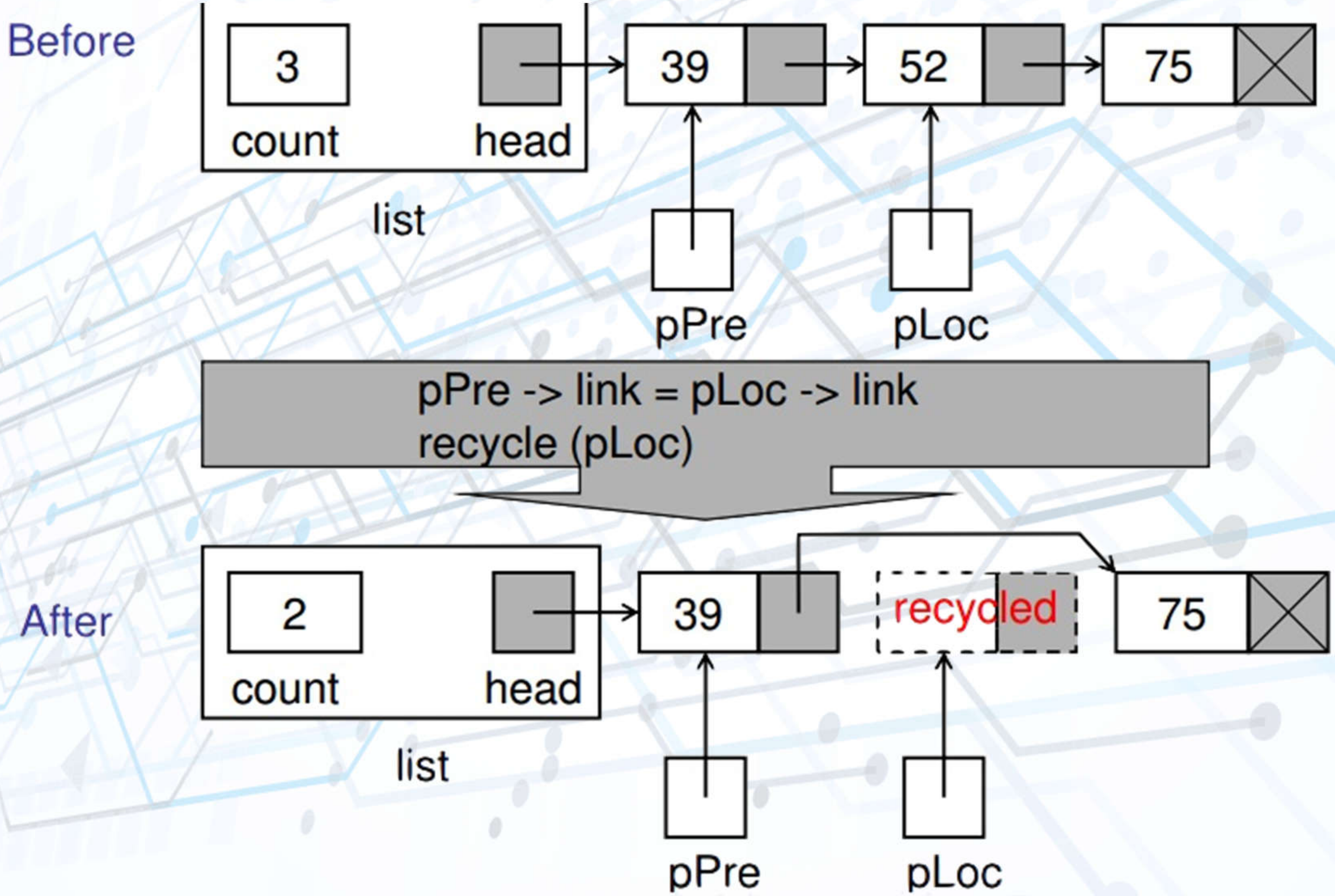
Delete a node from a linked list

1. Locate the pointer p in the list which points to the node to be deleted (pLoc will hold the node to be deleted).
   - If that node is the first element in the List: p is list.head.
   - Otherwise: p is pPre->link, where pPre points to the predecessor of the node to be deleted.
2. p points to the successor of the node to be deleted.
3. Recycle the memory of the deleted node.

Delete first node



General deletion case



Delete a node from a linked list

- Removal is successful when the node to be deleted is found.
- There is no difference between deleting the node from the beginning of the list and deleting the only node in the list.

```
list.head = pLoc->link
recycle(pLoc)
```

- There is no difference between deleting a node from the middle and deleting a node from the end of the list.

```
pPre->link = pLoc->link
recycle(pLoc)
```

Delete a node from a linked list

Algorithm deleteNode(ref list <metadata>,
val pPre <node pointer>,
val pLoc <node pointer>,
ref dataOut <dataType>)

Deletes data from a linked list and returns it to calling module.

Pre: list is metadata structure to a valid list
pPre is a pointer to predecessor node
pLoc is a pointer to node to be deleted
dataOut is variable to receive deleted data

Post: data have been deleted and returned to caller

Delete a node from a linked list

```
dataOut = pLoc -> data
if pPre = null then
    // Delete first node
    list.head = pLoc -> link
else
    // Delete other nodes
    pPre -> link = pLoc -> link
end
list.count = list.count - 1 recycle (pLoc)
return
End deleteNode
```

Delete a node from a linked list

```
template<class List_ItemType>
List_ItemType
LinkedList<List_ItemType>::DeleteNode(Node<List_ItemType>*pPre,Node<List_ItemType>*pLoc){
    List_ItemType result=pLoc->data;
    if(pPre==NULL){
        this->head=pLoc->link;
    }else{
        pPre->link=pLoc->link;
    }

    this->count--;
    delete pLoc;
    return result;
}
```

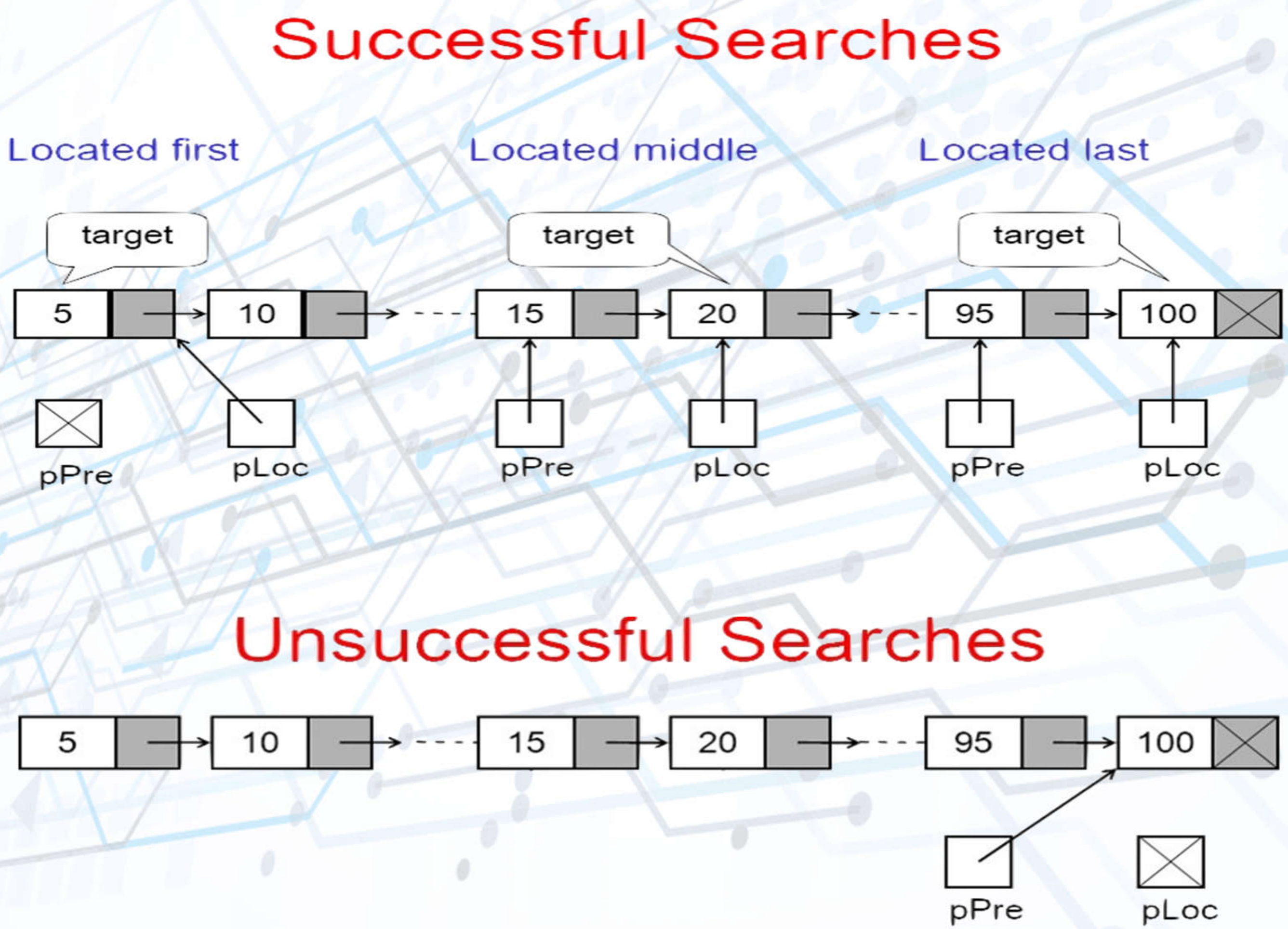
Searching in a linked list

- Sequence Search has to be used for the linked list.
- Function Search of List ADT:

```
<ErrorCode> Search (val target <dataType>,
                ref pPre <pointer>, ref pLoc <pointer>)
```

Searches a node and returns a pointer to it if found.

Searching in a linked list



Searching in a linked list

Algorithm Search(val target <dataType>,
ref pPre <node pointer>,
ref pLoc <node pointer>)

Searches a node in a singly linked list and return a pointer to it if found.

Pre: target is the value need to be found

Post: pLoc points to the first node which is equal target, or is NULL if not found.
pPre points to the predecessor of the first node which is equal target, or points to the last node if not found.

Return found or notFound

Searching in a linked list

```
pPre = NULL
pLoc = list.head
while (pLoc is not NULL) AND (target != pLoc ->data) do
    pPre = pLoc
    pLoc = pLoc ->link
end
if pLoc is NULL then
    return notFound
else
    return found
end
End Search
```

Searching in a linked list

```
template<class List_ItemType>
int LinkedList<List_ItemType>::Search(List_ItemType
value,Node<List_ItemType>*&pPre,Node<List_ItemType>*&pLoc){
    pPre=NULL;
    pLoc=this->head;
    while(pLoc!=NULL&&pLoc->data!=value){
        pPre=pLoc;
        pLoc=pLoc->link;
    }
    return(pLoc!=NULL);
    //found:1;notfound:0
}
```

Traverse a linked list

Traverse module controls the loop: calling a user-supplied algorithm to process data

Algorithm Traverse(ref <void> process (ref Data <DataType>))
Traverses the list, performing the given operation on each element.
Pre: process is user-supplied
Post: The action specified by process has been performed on every element in the list, beginning at the first element and doing each in turn.

pWalker = list.head
while *pWalker not null* **do**
  process(pWalker -> data)
  pWalker = pWalker -> link
end
End Traverse

Traverse a linked list

```
template<class List_ItemType>
void LinkedList<List_ItemType>::Traverse(){
    Node <List_ItemType>*p=head;
    while(p!=NULL){
        p->data++;//process data here!!!
        p=p->link;
        }
}

template<class List_ItemType>
void LinkedList<List_ItemType>::Traverse2(List_ItemType *&visit){
    Node <List_ItemType>*p=this->head;
    int i=0;
    while(p!=NULL && i<this->count){
        visit[i]=p->data;
        p=p->link;
        i++;
    }
}
```

Destroy a linked list

Algorithm destroyList (val list <metadata>)

Deletes all data in list.

Pre: list is metadata structure to a valid list

Post: all data deleted

```
while list.head not null do
    dltPtr = list.head
    list.head = this.head -> link
    recycle (dltPtr)
end
No data left in list. Reset metadata
list.count = 0
return
End destroyList
```

Destroy a linked list

```
template<class List_ItemType>
void LinkedList<List_ItemType>::Clear(){
    Node <List_ItemType>*temp;
    while(this->head!=NULL){
        temp=this->head;
        this->head=this->head->link;
        delete temp;
    }
    this->count=0;
}

template<class List_ItemType>
LinkedList<List_ItemType>::~LinkedList(){
    this->Clear();
}
```

Linked list implementation in C++

```
template<class List_ItemType>
class LinkedList{
  Node<List_ItemType>* head;
  int count;
protected:
  int InsertNode(Node<List_ItemType>* pPre,
                 List_ItemType value);
  List_ItemType DeleteNode(Node<List_ItemType>* pPre,
                           Node<List_ItemType>* pLoc);
  int Search(List_ItemType value,
             Node<List_ItemType>* &pPre,
             Node<List_ItemType>* &pLoc);

};
```

Linked list implementation in C++

```
template<class List_ItemType>
class LinkedList{
protected:
//...
public:
    LinkedList();
    ~LinkedList();
    void InsertFirst(List_ItemType value);
    void InsertLast(List_ItemType value);
    int InsertItem(List_ItemType value,int position);
    void DeleteFirst();
    void DeleteLast();
    int DeleteItem(int postion);
    int GetItem(int position,List_ItemType &dataOut);
    void Traverse();
    LinkedList<List_ItemType>*Clone();
    void Print2Console();
    void Clear();
};
```

Linked list implementation in C++

How to use Linked List data structure?

```
int main(int argc,char*argv[]){
    LinkedList<int>*myList=new LinkedList<int>();
    myList->InsertFirst(15);
    myList->InsertFirst(10);
    myList->InsertFirst(5);
    myList->InsertItem(18,3);
    myList->InsertLast(25);
    myList->InsertItem(20,3);
    myList->DeleteItem(2);
    cout<<"List 1:"<<endl;
    myList->Print2Console();
}
```

How to use Linked List data structure?

```
//...
int value;
LinkedList<int>*myList2=myList->Clone();
cout<<"List 2:"<<endl;
myList2->Print2Console();
myList2->GetItem(1,value);
cout<<"Value at position 1:"<<value;

delete myList;
delete myList2;
return 1;


}
```

Sample Solution: Insert

```
template<class List_ItemType>
int LinkedList<List_ItemType>::InsertItem(List_ItemType value,int position){
    if(position<0||position>this->count)
        return 0;
    Node<List_ItemType>*newPtr,*pPre;
    newPtr=new Node<List_ItemType>();
    if(newPtr==NULL)
        return 0;
    newPtr->data=value;
    if(head==NULL){
        head=newPtr;
        newPtr->link=NULL;
    }else if(position==0){
        newPtr->link=head;
        head=newPtr;
    }
```

Sample Solution: Insert

```
else{
    //Find the position of pPre
    pPre=this->head;

    for(int i=0;i<position-1;i++)
        pPre=pPre->link;

    //Insert new node
    newPtr->link=pPre->link;
    pPre->link=newPtr;
}

this->count++;
return 1;
}
```

Sample Solution: Delete

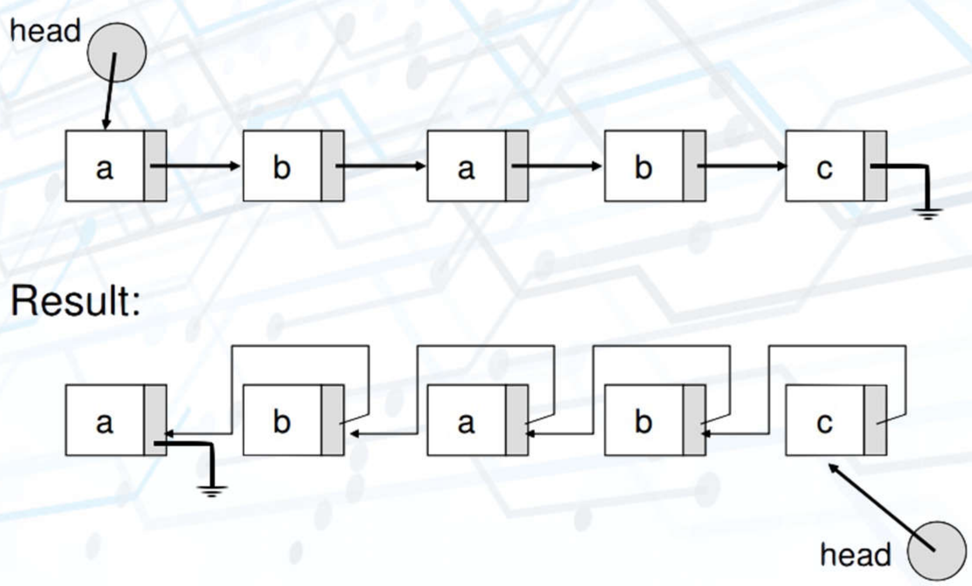
```
template<class List_ItemType>
int LinkedList<List_ItemType>::DeleteItem(int position){
    if(position<0||position>this->count)
        return0;
    Node <List_ItemType>*dltPtr,*pPre;
    if(position==0){
        dltPtr=head;
        head=head->link;
    }else{
        pPre=this->head;
        for(int i=0;i<position-1;i++)pPre=pPre->link;
            dltPtr=pPre->link;pPre->link=dltPtr->link;
    }
    delete dltPtr;
    this->count--;
    return 1;
}
```

Sample Solution: Clone

```
template<class List_ItemType>LinkedList<List_ItemType>*LinkedList<List_ItemType>::Clone(){
    LinkedList<List_ItemType>*result=new LinkedList<List_ItemType>();

    Node<List_ItemType>*p=this->head;

    while(p!=NULL){
        result->InsertLast(p->data);
        p=p->link;
    }

    result->count=this->count;
    return result;
}
```

Exercise

```
template <class List_ItemType>
void LinkedList<List_ItemType>::Reverse(){
  // ...
}
```



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.