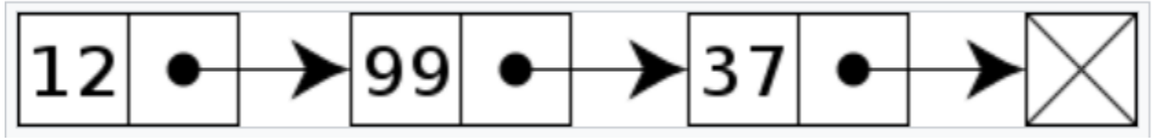
Data Structures and Algorithms

List ADT implementations:

Singly Linked List

Linked List

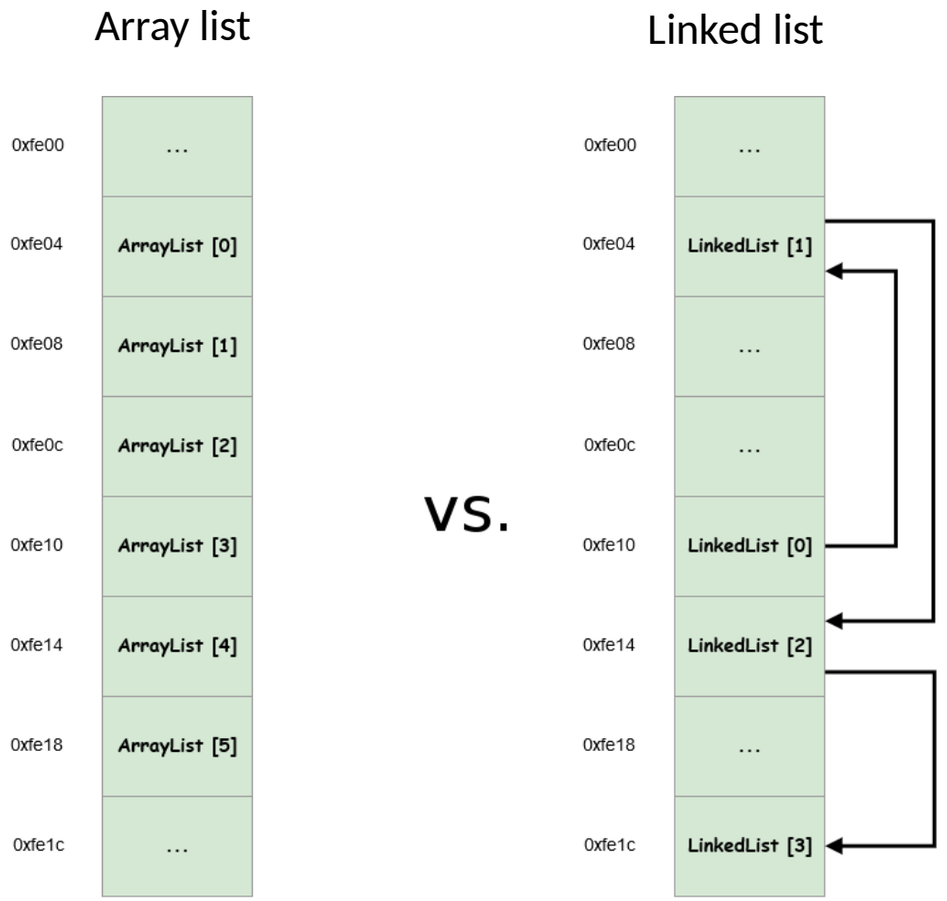


- In computer science, a linked list is a linear collection of data elements whose order is not given by their physical placement in memory. Instead, each element points to the next.
- It is a data structure consisting of a collection of nodes which together represent a sequence. In its most basic form, each node contains: data, and a reference (in other words, a link) to the next node in the sequence.
- This structure allows for efficient insertion or removal of elements from any position in the sequence during iteration.
- More complex variants add additional links, allowing more efficient insertion or removal of nodes at arbitrary positions.
- A drawback of linked lists is that access time is linear (O(n)). Faster access, such as random access, is not feasible. Arrays have better cache locality compared to linked lists.
- Linked lists are among the simplest and most common data structures. They can be used to implement several other common abstract data types, including lists, stacks, queues, associative arrays, and symbolic expressions.

https://en.wikipedia.org/wiki/Linked_list

Linked list and Array list in memory

- The principal benefit of a linked list over a conventional array is that the list elements can be easily inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk, while restructuring an array at run-time is a much more expensive operation.
- Linked lists allow insertion and removal of nodes at any point in the list, and allow doing so with a constant number of operations by keeping the link previous to the link being added or removed in memory during list traversal.

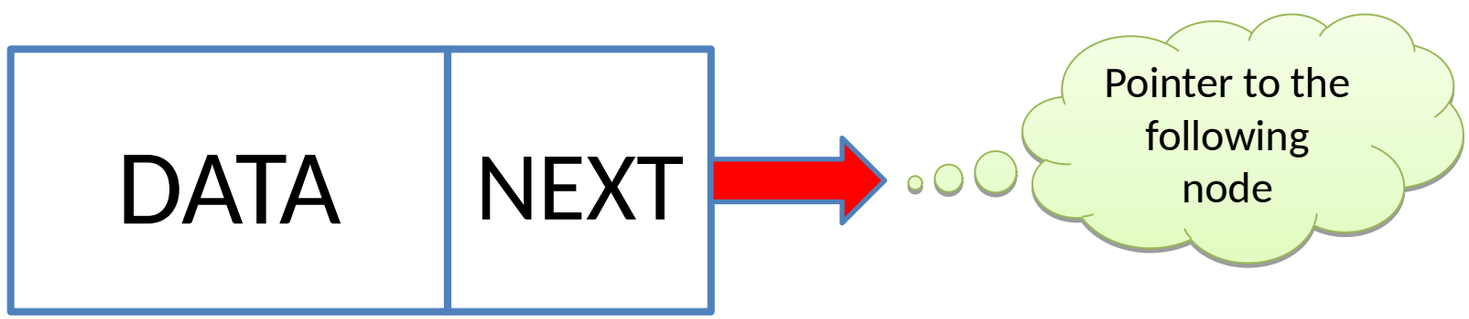


https://en.wikipedia.org/wiki/Linked_list

https://betterprogramming.pub/data-structures-whats-a-list-ca04b0ba9fa2

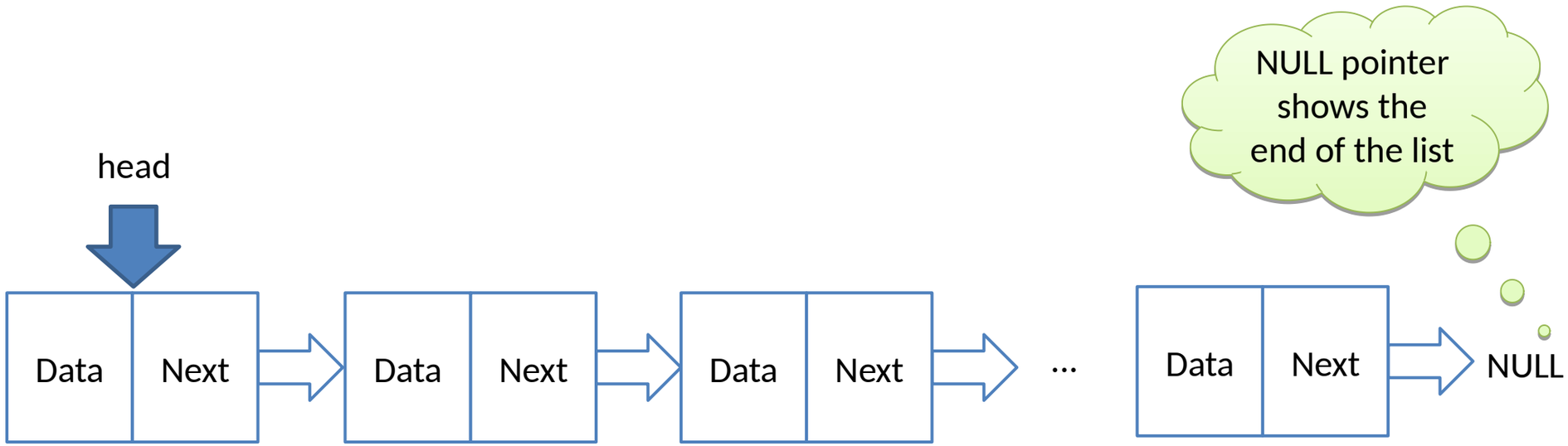
Structure of a Node

- The primary element of linked lists is nodes. When an element is added to the list, the element is internally stored on the node.
- The nodes of a singly-linked lists list contain a link to the next node.



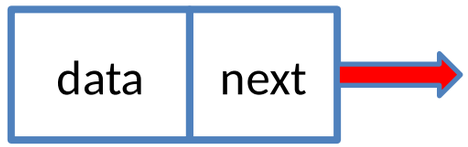
Singly linked list

- The list elements are linked to their neighbor element in a singly linked list.



Nodes in C++

- In a linked list, all connections are made using a pointer inside the node.
- Class or struct can be preferred to create nodes. In the node examples given below, the list data type is defined as an integer.
- It will then be expanded to objects or basic data types with the template class definition.

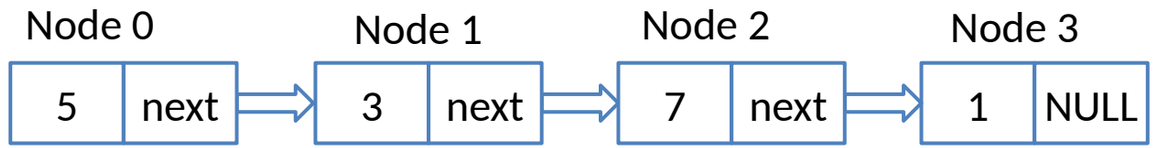



```
class Node{
public:
    int data;
    Node *next;
    Node(int data,Node
    *next=NULL){
        this->data=data;
        this->next=next;
    }
    };
```

```
struct Node{
    int data;
    Node *next;
    Node(int data,Node *next=NULL)
    {
        this->data=data;
        this->next=next;
    }
    };
```

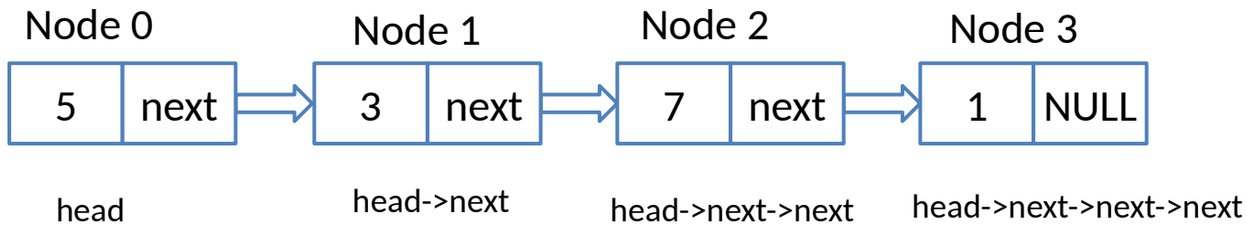
Building a simple linked list

```
//nodes
Node *node1=new Node(5);
Node *node2=new Node(3);
Node *node3=new Node(7);
Node *node4=new Node(1);

// connections
node1->next=node2;
node2->next=node3;
node3->next=node4;
node4->next=NULL;//the end of the
list
```



- In the example on the right, four nodes have been created and connected to a linked list.
- The NULL pointer shows the end of the list.
- Here, we may not be required to write this line because the ***next*** pointer is already defined as NULL by default in the Node class

Singly linked list basics

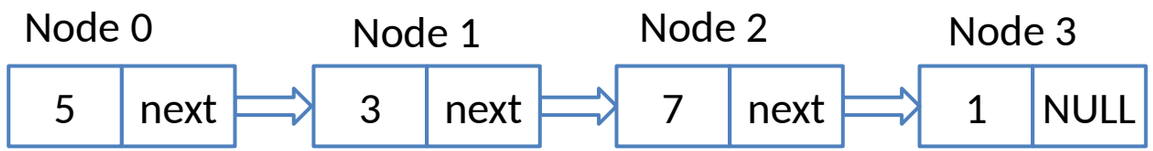



```
//nodes
Node *head;
head=new Node(5);
head->next=new Node(3);
head->next->next=new Node(7);
head->next->next->next=new Node(1);
head->next->next->next->next=NULL;
```

- Instead of defining a separate pointer for each node, only the first element of the list can be defined.

Singly linked list basics

```
//nodes
Node *head, *temp;
//first node is connected to head
head=new Node(5);

//set temp to the head of the list
temp=head;
//connect other elements using temp
//add an item
temp->next=new Node(3);
//move forward
temp=temp->next;
//add an item
temp->next=new Node(7);
//move forward
temp=temp->next;
//add an item
temp->next=new Node(11);
//move forward
temp=temp->next;
//temp=NULL;
```



- Adding elements becomes more practical if a temporary pointer is defined in addition to the first node.
- Here you go to the next node with **temp=temp->next**.

Travelling through linked list

```
//print list
//set temp to the head of the list
temp=head;
//print first item
cout<<temp->data<<endl;

//move forward
temp=temp->next;
//print the next list item
cout<<temp->data<<endl;

//move forward
temp=temp->next;
//print the next list item
cout<<temp->data<<endl;

//move forward
temp=temp->next;
//print the next list item
cout<<temp->data<<endl;

//move forward
temp=temp->next;
//print the next list item
cout<<temp->data<<endl;
```

- In order to print the elements of the list we created, the pointer defined with temp is set back to the first element.
- After an element is printed, **temp=temp->next** is also pointed to the next node.

Using while loop to travel through linked list

```
void print(Node *temp){
    while (temp!=NULL){
        cout<<temp->data<<endl;
        temp=temp->next;
    }
}
```

```
void print(Node *temp){
    if (temp!=NULL){
        cout<<temp->data<<endl;
        print(temp=temp->next);
    }
}
```

- The printing process can be defined as a function.
- The entire list is printed if the temp variable is selected as the first element when calling the function.
- Traveling over the items can also be done recursively.

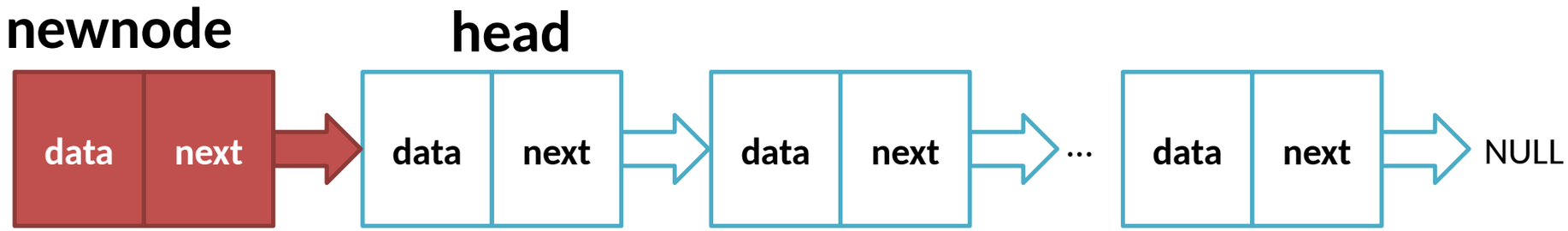
```
void yazdir(Node *temp){

    if  (temp != NULL) {
        cout << temp->data << endl;
        yazdir(temp->next);
    }

}
```

Singly linked list basics

```
int length(Node *temp){
int cnt=0;
    while (temp!=NULL){
        cnt++;
        temp=temp->next;
    }
    return cnt;
    }
```

- The elements are visited one by one, and the counter is increased by one for each element.
- We usually use a similar approach for reading, adding/deleting.

Adding an element to the front



- If the list is empty, the new element is assigned as the first element.
- If the list has one or more elements, the new element is located before the first element.
  - newnode->next=head;
  - head=newnode;

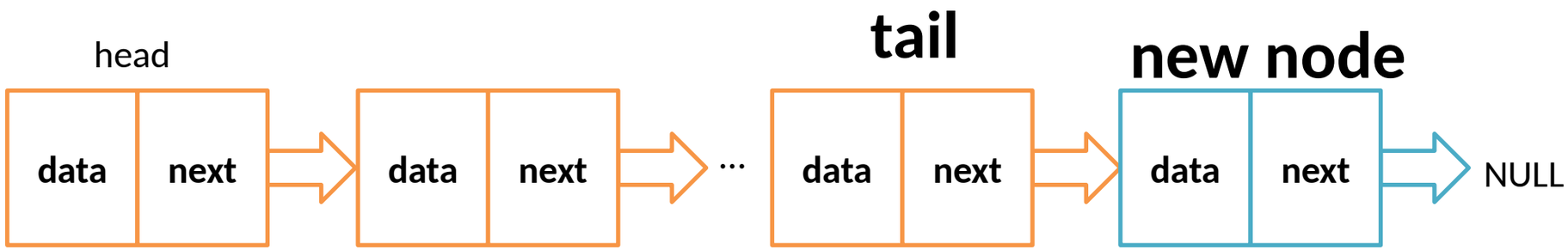
Adding an element to the front

```
void push_front(int data){
    Node *node=new Node(data);
    if (head==NULL){//if
    (head==0){
        head=node;
    }
    else{
        node->next=head;
        head=node;
    }
    }
```

```
//define a singly linked list
SinglyLinkedList list1;
//add elements
list1.push_front(10);
list1.push_front(8);
list1.push_front(7);
list1.push_front(5);
list1.push_front(11);
list1.push_front(3);
//the number of elements
cout<<"the number of elements:
"<<list1.length()<<endl;
//print list
list1.print();
```

```
<terminated> (exit value: 0) singlylinked_list1.exe [C/C++ Application] C:
the number of elements: 6
list elements:
        3       11      5       7       8       10
```

Adding an element to the end of the list



- If the list is empty, the new element is assigned as the first element.
- If there are one or more elements in the list, the last element of the list is detected.
- A new node is defined and connected to the last element.

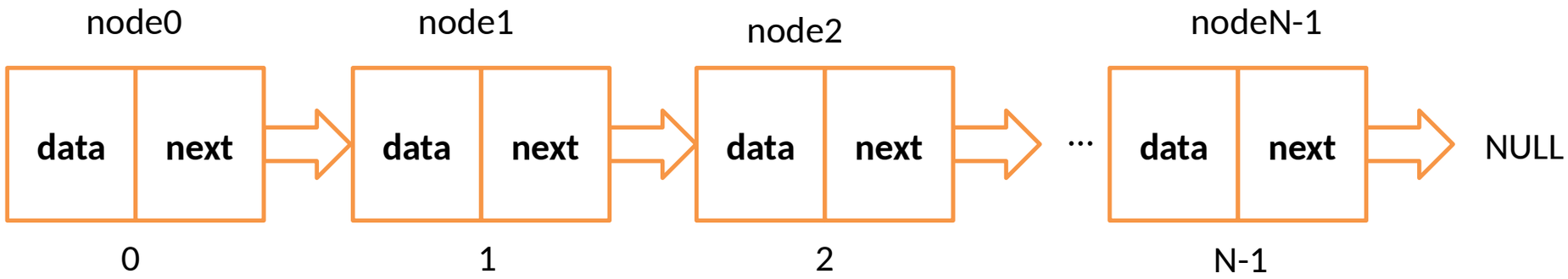
Adding an element to the end of the list

```
void push_back(int data){
    Node *node=new Node(data);
    if (head==NULL){
        head=node;
    }
    else{
        Node *temp=head;
        while(temp->next!=NULL){
            temp=temp->next;
        }
        temp->next=node;
    }
    }
```

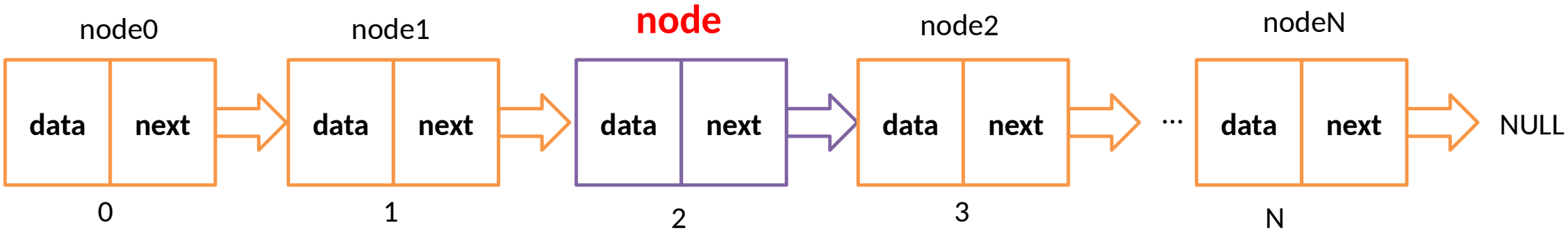
```
//define a singly linked
list
SinglyLinkedList list1;
//add elements
list1.push_back(10);
list1.push_back(8);
list1.push_back(7);
list1.push_back(5);
list1.push_back(11);
list1.push_back(3);
//the number of elements
cout<<"the number of
elements:
"<<list1.length()<<endl;
//print list
list1.print();
```

```
<terminated> (exit value: 0) singlylinked_list1.exe [C/C++ Application]
the number of elements: 6
list elements:
        10      8       7       5       11      3
```

Inserting an element



- **Node *node=new Node(veri);**
- **node->next=node1->next;**
- **node1->next=node;**



Inserting an element

```
Node* find_prev(int inx){
    Node *temp=head;
    int cnt=0;
    while(temp->next!=NULL){
        if (cnt==inx-1) break; //location
        found temp=temp->next;cnt++;
    }
        return temp;
    }
    void insert(int inx,int data){
    if (inx==0){
        push_front(data);
    }
    else{
        Node *node=new Node(data);
        Node *temp=find_prev(inx);
        node->next=temp->next;
        temp->next=node;
    }
    }
```

```
//define a singly linked
list
SinglyLinkedList list1;
//insert elements
list1.insert(0,10);
list1.print();
list1.insert(0,8);
list1.print();
list1.insert(1,7);
list1.print();
list1.insert(2,5);
list1.print();
list1.insert(2,11);
list1.print();
list1.insert(4,3);
list1.print();
```

```
<terminated> (exit value: 0) singlylinked_list1.exe [C/C++ Application] C:
list elements:
        10
list elements:
        8       10
list elements:
        8       7       10
list elements:
        8       7       5       10
list elements:
        8       7       11      5       10
list elements:
        8       7       11      5       3       10
```

Inserting an element

Exception definition for index

```
void insert(int inx,int data){

    if (inx<0 || inx>length()) throw exception();

    if (inx==0){
        push_front(data);
    }
    else{
        Node *node=new Node(data);
        Node *temp=find_prev(inx);
        node->next=temp->next;
        temp->next=node;
    }
    }
```

```
//define a singly linked list
SinglyLinkedList list1;

//insert elements
list1.insert(0,10);
list1.print();
list1.insert(0,8);
list1.print();
list1.insert(1,7);
list1.print();
list1.insert(2,5);
list1.print();
list1.insert(2,11);
list1.print();
list1.insert(40,3);
list1.print();
```

location not valid

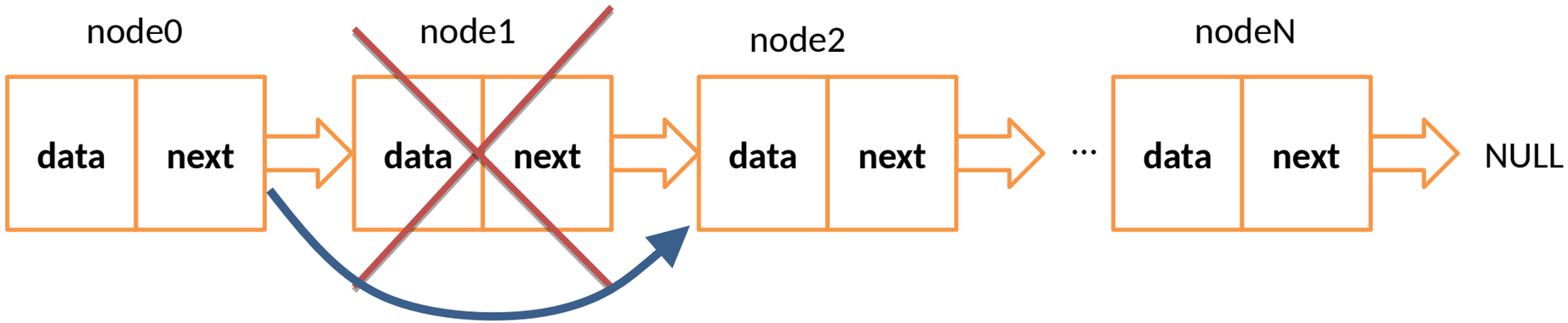
```
<terminated> (exit value: 3) singlylinked_list1.exe [C/C++ Application] C:\DATASTRUC\wo
list elements:
        10
list elements:
        8       10
list elements:
        8       7       10
list elements:
        8       7       5       10
list elements:
        8       7       11      5       10
terminate called after throwing an instance of 'std::exception'
  what():  std::exception
```

Inserting an element

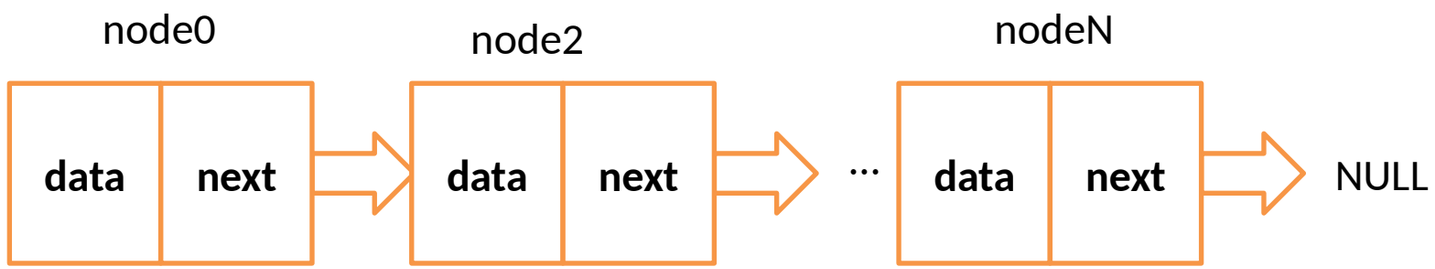
```
SinglyLinkedList list1;
try {
    //insert elements
    list1.insert(0, 10);
    list1.print();
    list1.insert(0, 8);
    list1.print();
    list1.insert(1, 7);
    list1.print();
    list1.insert(2, 5);
    list1.print();
    list1.insert(2, 11);
    list1.print();
    list1.insert(40, 3);
    list1.print();
    } catch (exception &e) {
        cout << e.what() << endl;
    }
```

```
<terminated> (exit value: 0) singlylinked_list1.exe [C/C++ Appli
list elements:
        10
list elements:
        8       10
list elements:
        8       7       10
list elements:
        8       7       5       10
list elements:
        8       7       11      5       10
std::exception
```

Removing an elemnet



- **node0->next=node1->next;**
- **delete node1;**



Removing an element

```
void remove(int inx){
    if (inx < 0 || inx >= length()) throw exception();
        Node * temp=head;
        if (inx==0){
            head=head->next;
        delete temp;
    }
    else{
        Node *temp = find_prev(inx);
        Node *removenode=temp->next;
        temp->next=removenode->next;
        delete removenode;
    }
    }
```

```
list1.insert(0, 10);
list1.insert(0, 8);
list1.insert(1, 7);
list1.insert(2, 5);
list1.insert(2, 11);
list1.insert(4, 3);
list1.print();

list1.remove(0);
list1.print();
list1.remove(3);
list1.print();
list1.remove(3);
list1.print();
```

```
<terminated> (exit value: 0) singlylinked_list1.exe [C/C++ Application] C
list elements:
        8       7       11      5       3       10
list elements:
        7       11      5       3       10
list elements:
        7       11      5       10
list elements:
        7       11      5
```

Reading an element

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

    }
```

```
SinglyLinkedList list1;
try {
     //insert elements
     list1.insert(0, 10);
     list1.insert(0, 8);
     list1.insert(1, 7);
     list1.insert(2, 5);
     list1.insert(2, 11);
     list1.insert(4, 3);
     list1.print();

     int i;
     for (i = 0; i < list1.length(); +
     +i) {
         cout<<list1.at(i)<<endl;
     }

     } catch (exception &e) {
          cout << e.what() << endl;

     }
```

Reading an element

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

    }
```

```
SinglyLinkedList list1;
try {
     //insert elements
     list1.insert(0, 10);
     list1.insert(0, 8);
     list1.insert(1, 7);
     list1.insert(2, 5);
     list1.insert(2, 11);
     list1.insert(4, 3);
     list1.print();

     int i;
     for (i = 0; i < list1.length(); +
     +i) {
         cout<<list1.at(i)<<endl;
     }

     } catch (exception &e) {
          cout << e.what() << endl;

     }
```

Reading an element

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

- If there is no const definition in the function, we can assign the element as follows.

```
//print the item at 3
cout<<list1.at(3)<<endl;

//change the item at 3
list1.at(3)=1234;

//print the item at 3
cout<<list1.at(3)<<endl;
```

If we do not want such a assignment, we can prevent this by writing const at the beginning of the function.

```
const int &at(int inx){..
```

clearing the list

```
void clear(){
    Node *temp=head;
    Node *delnode;

    while(temp!=NULL){
        delnode=temp;
        temp=temp->next;
        delete delnode;
    }
    head=NULL;
    }
```

Delete nodes one by one

List is completely empty. Set head to NULL

```
~SinglyLinkedList() { //destructor
    //delete nodes before destructing the list
    clear();
}
```