DET102 Data Structures and Algorithms

Lecture 02: Vector and List

Abstract Data Type

- We want to know whether two cups can hold the same amount of water
- What to do ?
  - Calculate the volume by mathematical formulas
  - Fill cup 1 by water, pour the water to cup 2, overflow? vice versa
  - …
- We only care about the **result**, not how to get the result
- **Abstract Data Type** for cups!

Abstract Data Types

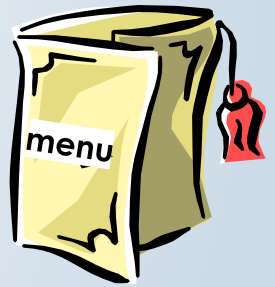

```
/*Octopus.h*/
class Octopus
{
private:
float value;
Person p;
Credit_Card_No n;
float Rewards;
…

public:
void Increase_value (..);
void Increase_credit(..);
void Pay_Transaction(..);
bool Identity_Verify(..);
Void accumulate();
…
};
```

Some advanced data types are very useful.

People tend to create modules that group together the data types and the functions for handling these data types. (~ Modular Design)

They form very useful toolkits for programmers.

When one search for a "tool", he looks at **what** the tools can do. i.e. He looks at the abstract data types (ADT).



He needs not know **how** the tools have been implemented.

ADT Example in C++: Set

Mathematically a set is a collection of items not in any particular order.

- **Value:**
  - A set of elements

  Condition: elements are distinct.
- **Operations for Set *s:**

1. void Add(ELEMENT e)

postcondition: e will be added to *s

2. void Remove(ELEMENT e)

precondition: e exists in *s

postcondition: e will be removed from *s

3. int Size()

postcondition: the no. of elements in *s will be returned.

…

- An **ADT** is a package of the declarations of a data type and the operations that are meaningful to it.
- We encapsulate the data type and the operations and hide them from the user.
- ADTs are implementation independent.

Abstract Data Type

The **set** ADT:

Consists of 2 parts:

1. Definition of values

   involves

   - definition
   - condition (optional)

2. Definition of operations

   each operation involves

   - header
   - precondition (optional)
   - postcondition

Value:

- A set of elements

  Condition: elements are distinct.

Operations for Set *s:

1. **void Add(ELEMENT e)**

   postcondition: e will be added to *s

2. **void Remove(ELEMENT e)**

   precondition: e exists in *s

   postcondition: e will be removed from *s

3. **int Size()**

   postcondition: the no. of elements in *s will be returned.

…

ADT and Data Structure

- In computer science, an abstract data type (ADT) is a mathematical model for data types.
- An abstract data type is defined by its behavior from the point of view of a user, of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.
- This mathematical model contrasts with data structures, which are concrete representations of data, and are the point of view of an implementer, not a user.

Basic Data Structures

- Linear data structures
  - Vector
  - List
  - Deque
  - Stack
  - Queue
  - Priority queue
  - Set
  - Multiset
  - Map
  - Multimap
- Nonlinear data structures
  - Tree
  - graph

Linear List

- A *Linear List* (or a *list*, for short)
  - is a sequence of n nodes x_1, x_2, .., x_n whose essential structural properties involve only the relative positions between items as they appear in a line.
- A list can be implemented as
  - array: statically allocated or dynamically allocated
  - linked list: dynamically allocated
- A list can be sorted or unsorted

Array

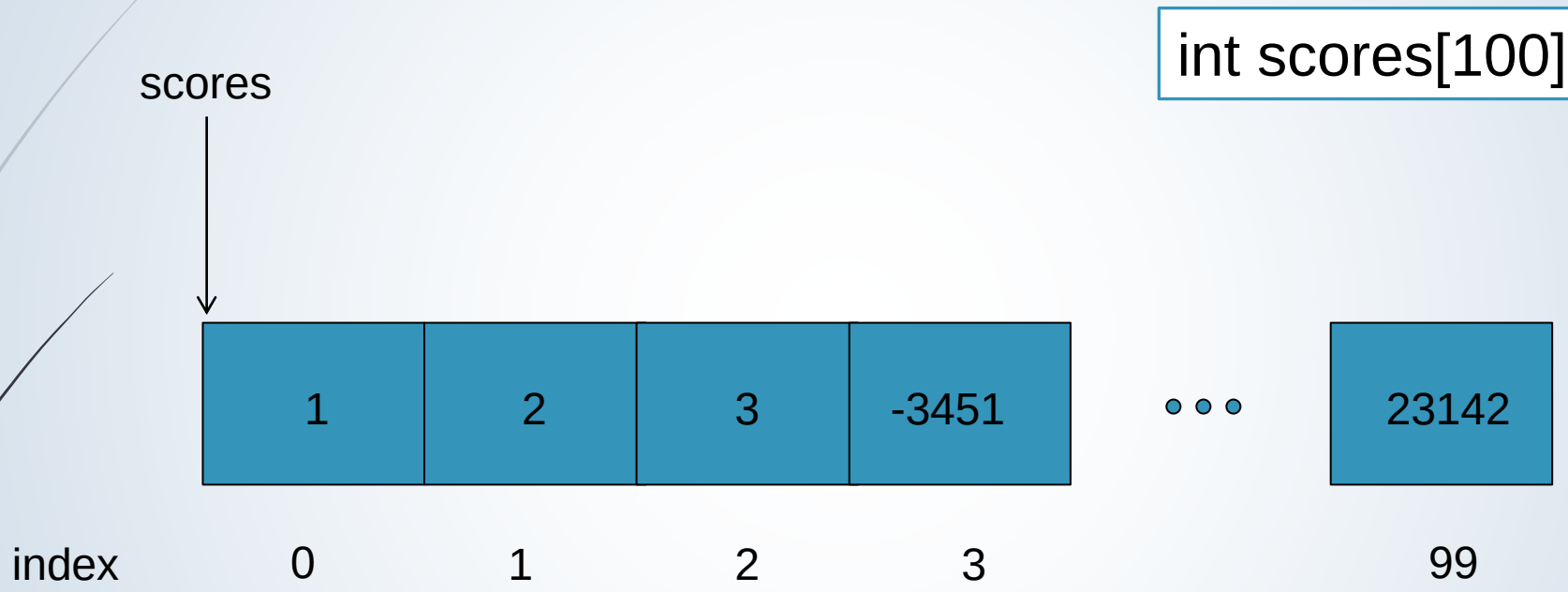
In many programming languages, array is the built-in data type.

For example, in C/C++ or Java, each array has a fixed size once it is created, and each array contains the same type of elements.

Must be a constant.

```
void ArrayTest(){
    int scores[100];
//operate on the elements of the scores array
    scores[0]=1;
    scores[1]=2;
    scores[2]=3;
}
```

Array in memory



An index is the number associated with the place in the array. We can easily access the specific element using its index, for example, the middle one.

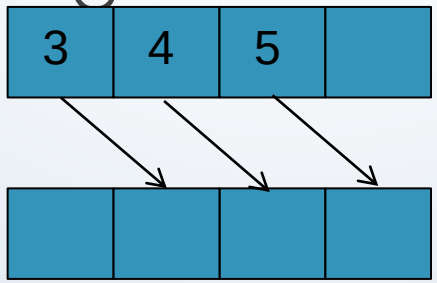
Disadvantages of Arrays

- The size of array is fixed. Inserting at the end may cause overflow.
  - this size is specified at compile time

For convenience, allocate "large enough "
- waste space: only 20 – 30 elements
- crash: more than 100 elements

- Deleting or inserting new elements at the front is expensive.

Insert 2



Inefficient!

How to improve array?

Vector (also called Array List)

- Access each element using a notion of index in [0,n-1]
- Index of element e = the number of elements that are before e
- It is a more general ADT than array.
- One of the famous containers in c++ Standard Template Library.
- Can store a sequence of objects.
- An element can be accessed, inserted or removed by specifying its index.

ADT of Vector

- Main methods:
  - at(i): return the element at index i without removing it
  - set(i,o): replace the element at index i with new value o
  - insert: insert a new element at given index
  - erase: remove element at some index
- Additional methods"
  - size( )
  - empty( )

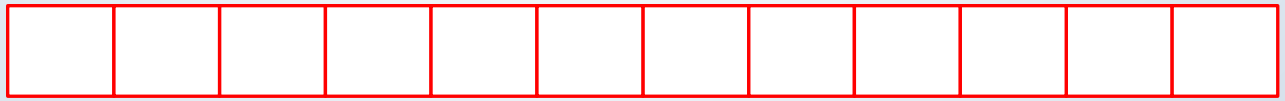
Reference: full list of member functions of Vector in STL

Applications of Vector

- Direct applications
  - sorted collection of objects
- Indirect applications
  - auxiliary data structure for algorithms
  - component of other data structures: e.g., stack
- Every place where you can use array

Array-based implementation of Vector

- Use an array A of size N
- A variable n keeps track of the size of the array list (number of elements stored)
- Operation at(i) is implemented in O(1) time by returning A[i]
- Operation set(i,o) is implemented in O(1) time by performing A[i] = o
- Operation insert(i, o): we need to make room for the new element by shifting forward the n - i elements A[i], …, A[n - 1] In the worst case (i = 0), this takes O(n) time
- Operation erase(i): we need to fill the hole left by the removed element by shifting backward the n - i - 1 elements A[i + 1], …, A[n - 1] In the worst case (i = 0), this takes O(n) time



Performance

- In the array-based implementation of vector:
  - The space used by the data structure is O(n)
  - size, empty, at and set run in O(1) time
  - insert and erase run in O(n) time in worst case
- If we use the array in a circular fashion, operations insert(0, x) and erase(0, x) run in O(1) time
- In an insert operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one.

Growable array-based vector

- In an insert(o) operation (without an index), we always insert at the end
- When the array is full, we replace the array with a larger one
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size
- For size n array, "expand" operation requires n copies

Which is better? Incremental or Doubling

- Comparison Method 1
  - Given the current size of S = n
  - Worst-case running time
    - Incremental strategy: O(1)
    - Doubling strategy: O(n)
- Are you happy?
  - Happy if your focus is really the worst-case
  - Unhappy
    - For doubling strategy, the total number of resizing array size would be small
- Can we reconsider the analysis method?

Which is better? Incremental or Doubling

- Comparison Method2
  - Compute the total time T(n) needed to perform a series of n insert(o) operations
  - Assume that we start with an empty stack represented by an array of size 1
- We call amortized time of an insert operation the average time taken by an insert over the series of operations, i.e., T(n)/n
  - This can be a fairer comparison in some cases
- Amortized analysis (均攤分析/平攤分析)

Amortized Analysis

- size(n): the number of elements stored in the vector after n insertion operations
- capacity(n): the capacity of the vector after n insertion operations
- T(n): the time spent on expanding for n consecutive insertion operations.
- We assume that there are already N elements in the vector before insertion operations.

- size(n)=N+n
- size(n)<= capacity(n) < 2*size(n)
- because N is a constant
- capacity(n)=O(size(n))=O(n)
- T(n)=aN+4N+8N+16N+….+ capacity(n) < 2* capacity(n)=O(n)
- amortized complexity for single insertion operation is O(1)

Expand()

```
template <typename T> void Vector<T>::expand() {
    if ( _size < _capacity )
        return;
    if ( _capacity < DEFAULT_CAPACITY )
        _capacity = DEFAULT_CAPACITY;
    T* oldElem = _elem;
    _elem = new T[_capacity <<= 1];
    for ( Rank i = 0; i < _size; i++ )
        _elem[i] = oldElem[i];
    delete [] oldElem;
}
```

Incremental Strategy Analysis

- We replace the old array with a new one k = n/c times
- The total time T(n) of a series of n insert operations is proportional to

  n + c + 2c + 3c + 4c + … + kc =

  n + c(1 + 2 + 3 + … + k) =

  n + ck(k + 1)/2
- Since c is a constant, T(n) is O(n + k^2), i.e., O(n^2)
- The amortized time of an insert operation is O(n)

Doubling Strategy Analysis

- We replace the old array with a new one $k = \log_2 n$ times
- The total time T(n) of a series of n insert operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k = n + 2^{k+1} - 1 = 3n - 1$$

- T(n) is O(n)
- The amortized time of an insert operation is O(1)

Vector: constructor

- Empty constructor
- Fill constructor
- Range constructor
- Copy constructor

```cpp
// constructing vectors
#include <iostream>
#include <vector>

int main ()
{
  // constructors used in the same order as described above:
  std::vector<int> first;                                // empty vector of ints
  std::vector<int> second (4,100);                       // four ints with value 100
  std::vector<int> third (second.begin(),second.end());  // iterating through second
  std::vector<int> fourth (third);                       // a copy of third

  // the iterator constructor can also be used to construct from arrays:
  int myints[] = {16,2,77,29};
  std::vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

  std::cout << "The contents of fifth are:";
  for (std::vector<int>::iterator it = fifth.begin(); it != fifth.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Vector: iterator

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin	Return const_iterator to beginning (public member function)
cend	Return const_iterator to end (public member function)
crbegin	Return const_reverse_iterator to reverse beginning (public member function)
crend	Return const_reverse_iterator to reverse end (public member function)

Vector: capacity

O(1)	size	Return size (public member function)
	max_size	Return maximum size (public member function)
	resize	Change size (public member function)
	capacity	Return size of allocated storage capacity (public member function)
O(1)	empty	Test whether vector is empty (public member function)
	reserve	Request a change in capacity (public member function)
	shrink_to_fit	Shrink to fit (public member function)

Vector: Element Access

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
data	Access data (public member function)

Vector::at

Returns a reference to the element at position n in the vector.

The function automatically checks whether n is within the bounds of valid elements in the vector, throwing an out_of_range exception if it is not (i.e., if n is greater than, or equal to, its size). This is in contrast with member operator[], that does not check against bounds.

```
1  // vector::at
2  #include <iostream>
3  #include <vector>
4
5  int main ()
6  {
7    std::vector<int> myvector (10);   // 10 zero-initialized ints
8
9    // assign some values:
10   for (unsigned i=0; i<myvector.size(); i++)
11     myvector.at(i)=i;
12
13   std::cout << "myvector contains:";
14   for (unsigned i=0; i<myvector.size(); i++)
15     std::cout << ' ' << myvector.at(i);
16   std::cout << '\n';
17
18   return 0;
19 }
```

Vector::front

Returns a reference to the first element in the vector.

Unlike member vector::begin, which returns an iterator to this same element, this function returns a direct reference.

Calling this function on an empty container causes undefined behavior.

```
// vector::front
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;

  myvector.push_back(78);
  myvector.push_back(16);

  // now front equals 78, and back 16

  myvector.front() -= myvector.back();

  std::cout << "myvector.front() is now " << myvector.front() << '\n';

  return 0;
}
```

Vector: Modifier

	assign	Assign vector content (public member function)
O(1)	push_back	Add element at the end (public member function)
O(1)	pop_back	Delete last element (public member function)
O(n)	insert	Insert elements (public member function)
O(n)	erase	Erase elements (public member function)
O(1)	swap	Swap content (public member function)
	clear	Clear content (public member function)
	emplace	Construct and insert element (public member function)
	emplace_back	Construct and insert element at the end (public member function)

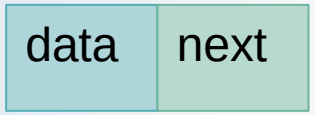
https://cplusplus.com/reference/vector/vector/

Linked List

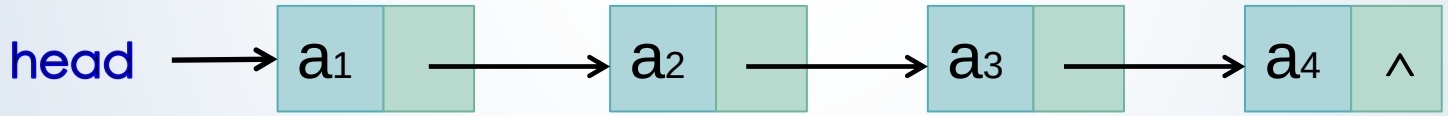
- Singly linked list
- Doubly linked list
- Circular Linked List

Singly Linked List

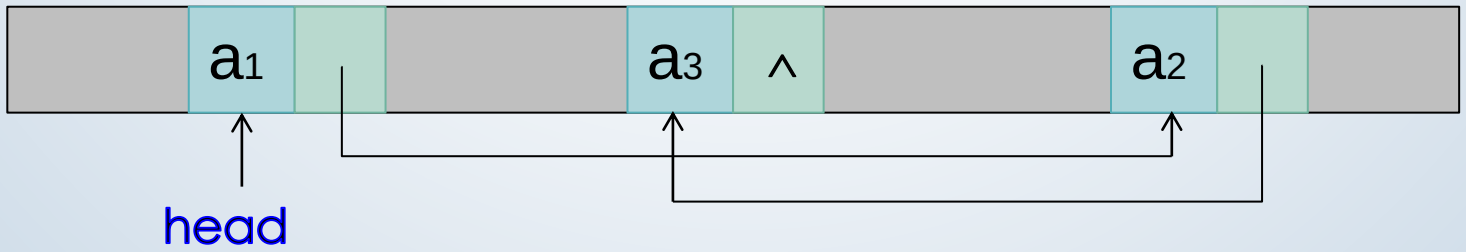
- Each item in the linked list is a node.



- Linear Structure



- Node can be stored in memory consecutively /or not (logical order and physical order may be different)



Linked List Types in C

- Node

```
struct node{
    int   data;
    struct node * next;
};
```

a pointer to the next node in the list

- Linked List

```
struct  linkedlist{
    struct node * head;
} ;
```

a pointer through which the list can be accessed.

Linked List Types in C++


```
// List.h
#include <string>
using namespace std;

    class ListNode
    {
    public:
        ListNode( int );
        ListNode( int, ListNode *);
        ListNode *get_Next()
        {
            return next;
        }
        …
    private:
        int data;
        ListNode *next;
    };
```

```
    class List
    {
    public:
        List( String  );
        List();
       //various member functions
    private:
        ListNode *first;
        String name;
    }
```

Linked List Type in Python

```
class Node:
    def __init__(self, value=None):
        self.value = value
        self.next = None
```

```
class LinkedList:
    def __init__(self):
    self.head = None
```

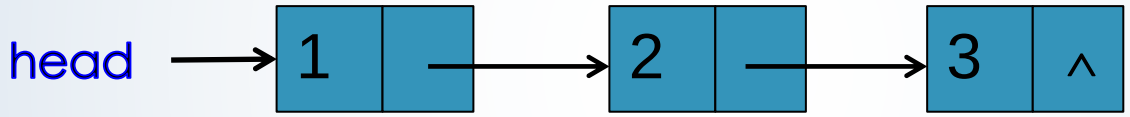
```
list1 = LinkedList()
list1.head = Node("Mon")

e2 = Node("Tue")
e3 = Node("Wed")

# Link first Node to second node
list1.head.next = e2

# Link second Node to third node
e2.next = e3
```

Operations on Linked List

- Create a linked list
- Get the length of the linked list
- Check the empty condition
- Traverse a linked list
- Insert a node in a linked list
  - At the front
  - In a sorted linked list
- Delete a node in a linked list
  - Search a node

Create a linked list

- Build a linked list {1,2,3}



How to create a simple linked list with three nodes?

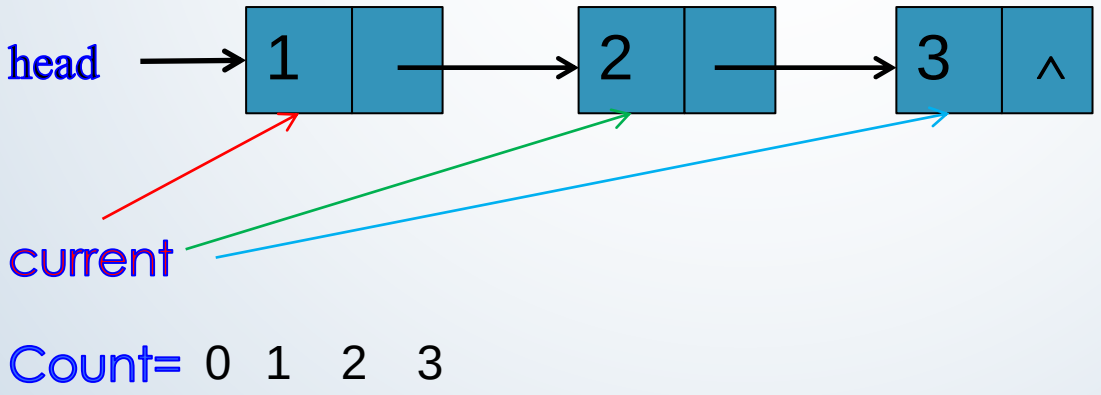
Step 1: create three nodes

Step 2: link these nodes together

Step 3: return the head pointer

Length() Function

- The Length() function takes a linked list and computes the number of elements in the list.
- Length() is a simple linked list function, but it demonstrates several concepts which will be used in later, more complex list functions...



PrintList() function

- Very similar to Length() function.
- Used to *traverse the linked list*.

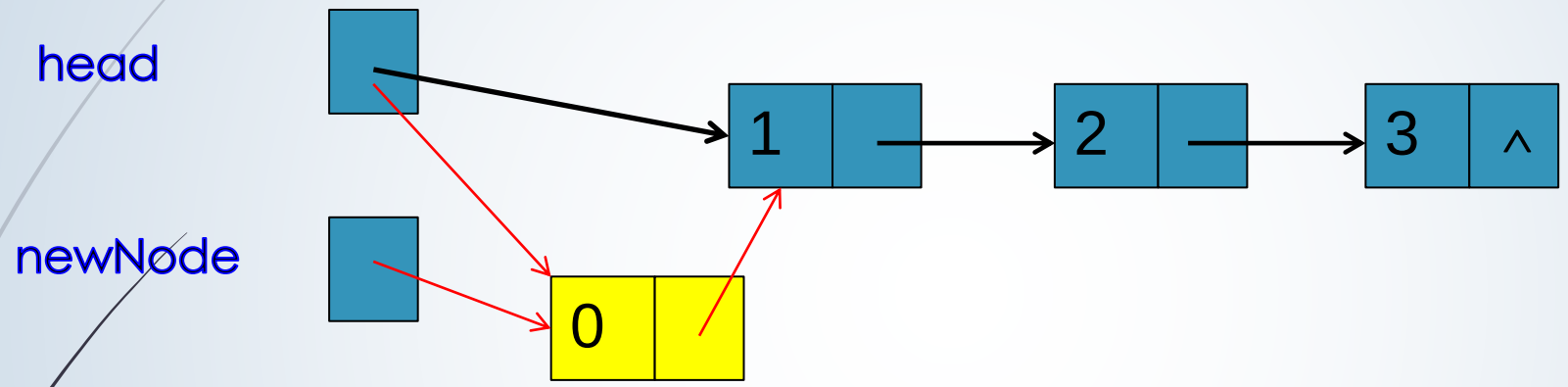
```
# Print the linked list
def listprint(self):
    printval = self.head
    while printval: # while printval is not None
        print (printval.value)
        printval = printval.next
```

Insert a node to a linked list

- Three cases when inserting a node
  - Insert a node at the font
  - Insert a node at the end of the linked list
  - Insert a node in the middle

One important case missing: Empty List

Insert at the front




```
def insertAtFrong(self,newdata):
    NewNode = Node(newdata)
    # Update the new nodes next to existing node
    NewNode.next = self.head
    self.head = NewNode
```

InsertAtFront()

Also called
Push()

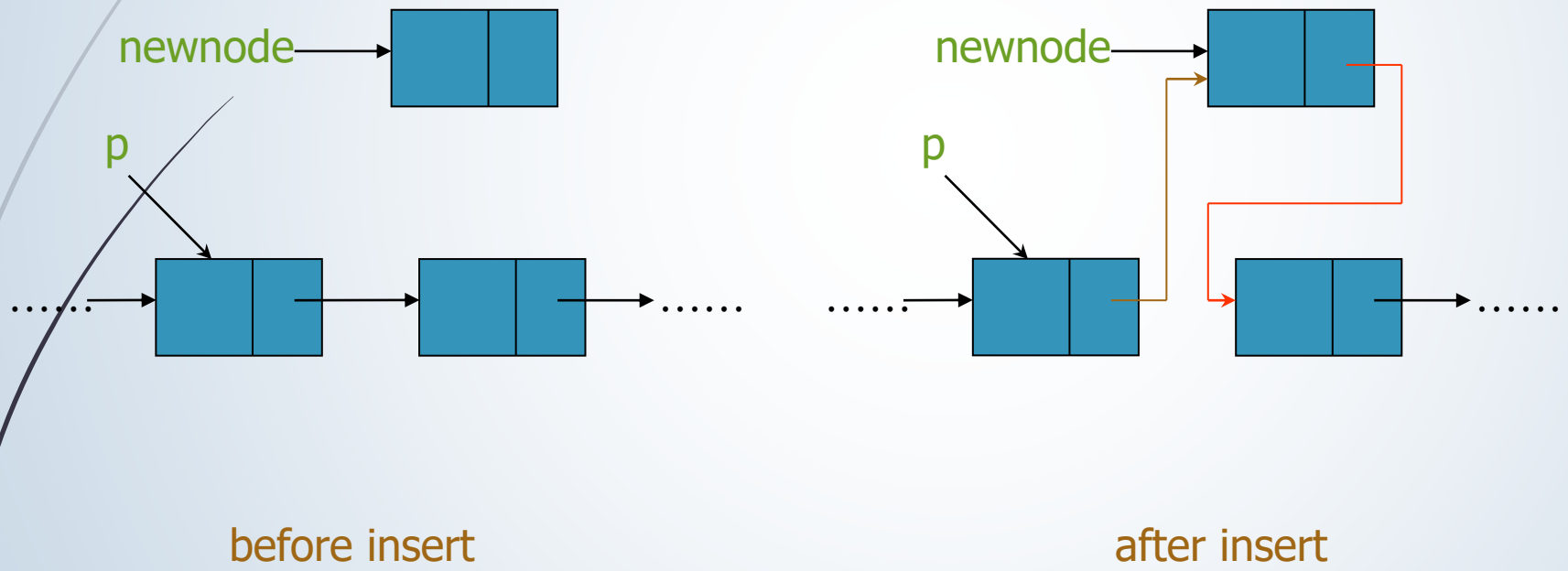
- Add a single node to the head end of any list.
  - Historically, it is called Push().
  - Alternately, it could be called InsertAtFront()
- Pay attention to the passing parameters.

Insert at the End

- This involves pointing the next pointer of the current last node of the linked list to the new data node. So the current last node of the linked list becomes the second last data node and the new node becomes the last node of the linked list.

```
# Function to add newnode
def insertAtEnd(self, newdata):
    NewNode = Node(newdata)
    if self.head is None:
        self.head = NewNode
        return
    laste = self.head
    while(laste.next):
        laste = laste.next
    laste.nextval=NewNode
```

Insert in the middle



before insert

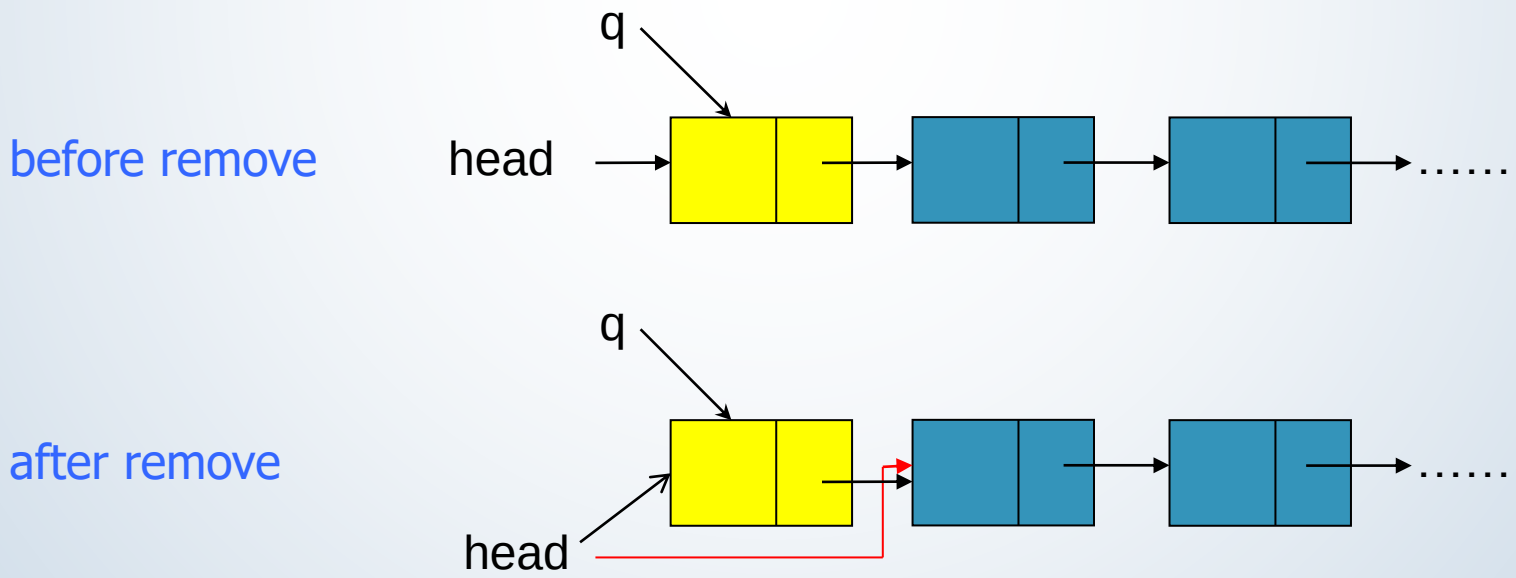
after insert

Remove a node

- Some data become useless and we want to remove them, how?
  - Search the useless node by data value
  - Remove this node
- We will encounter two cases
  - Removing a node at the beginning of the list
  - Removing a node **NOT** at the beginning of the list

Remove a node

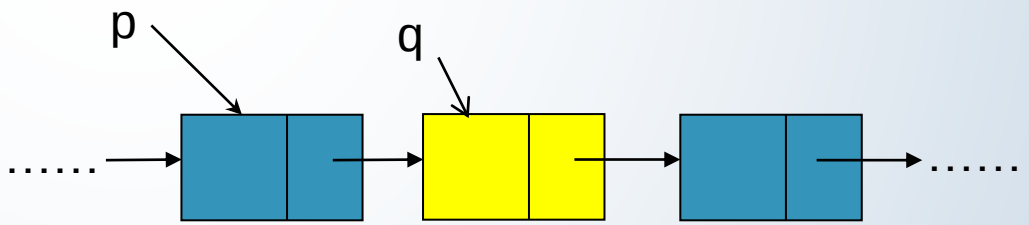
- Case 1: Remove a node at the beginning of the list
  - current status: the node pointed by "head" is unwanted.
  - The action we need: q=head; head=q.next;



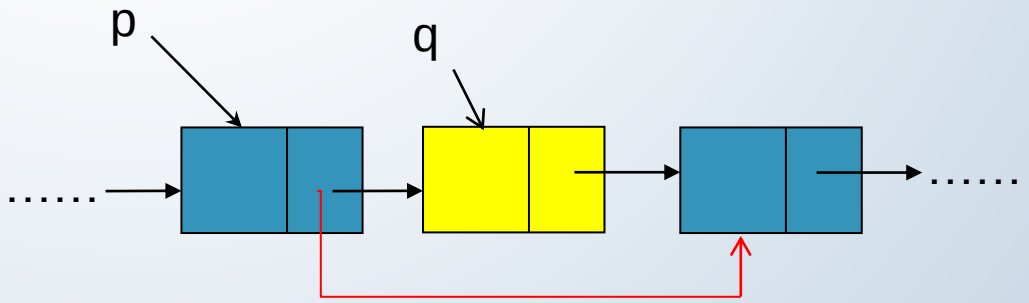
Remove a node

- Case 2: Remove a node not at the beginning of the list
  - Current status: q == p.next and the node pointed by q is unwanted
  - The action we need: p.next=q.next

before remove



after remove



Dummy Header Node

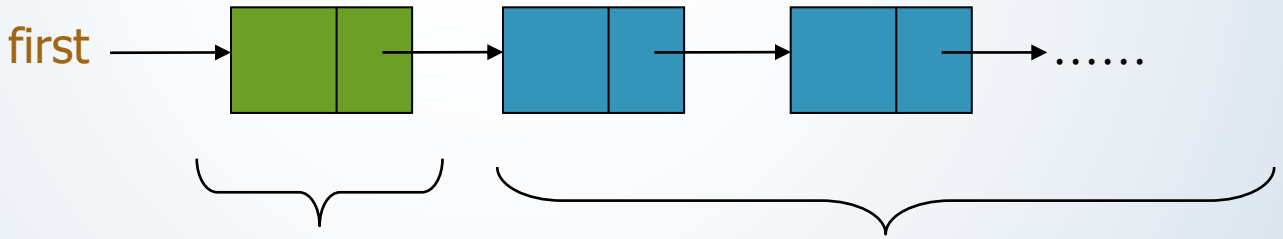
So many cases with Insert and Remove operations

We want simpler implementations!

What are the special cases? Why are they different?

One way to simplify:

- keep an extra node at the front of the list

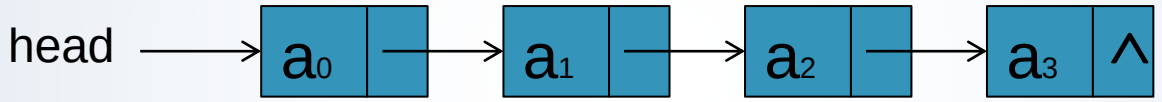


We don't care about the value of this node.

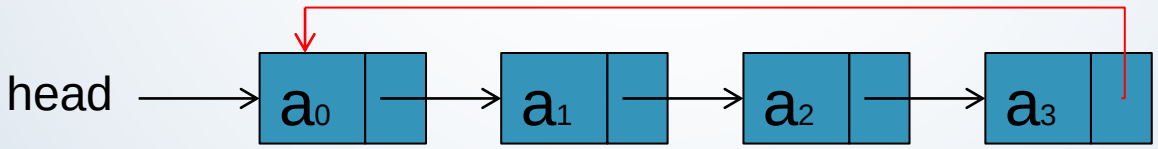
The value of these nodes are our data

Circular Lists

- Suppose that we are at node a_3 and want to reach node a_0, how?

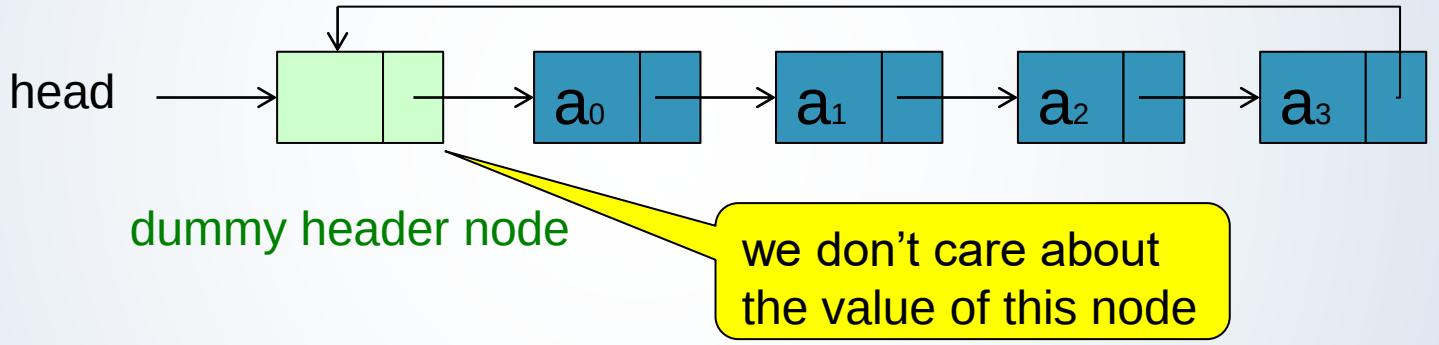


- If one extra link is allowed:

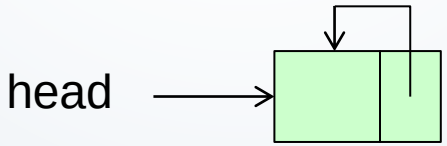


Circular Lists

- Dummy header node can also be added to make the implementation easier.

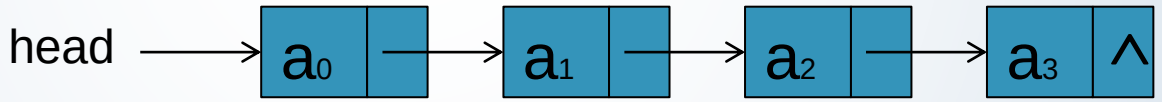


- Empty list

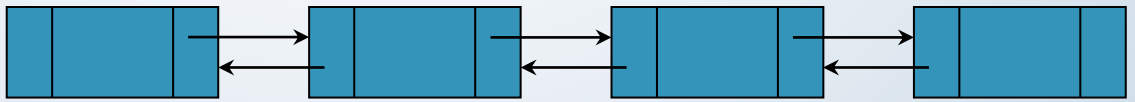


Doubly linked lists

- Problem with singly linked list
  - When at a_3, we want to get a_2
  - When deleting node a_3, we need to know the address of node a_2
  - When at a_3, it is difficult to insert between a_2 and a_3

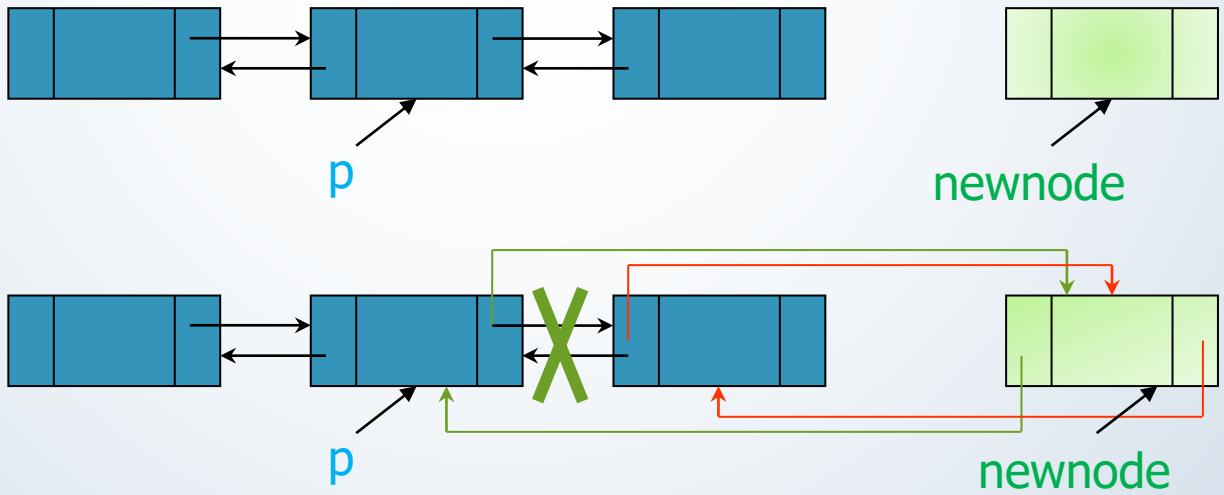


- If allowed to use more memory spaces, what to do?



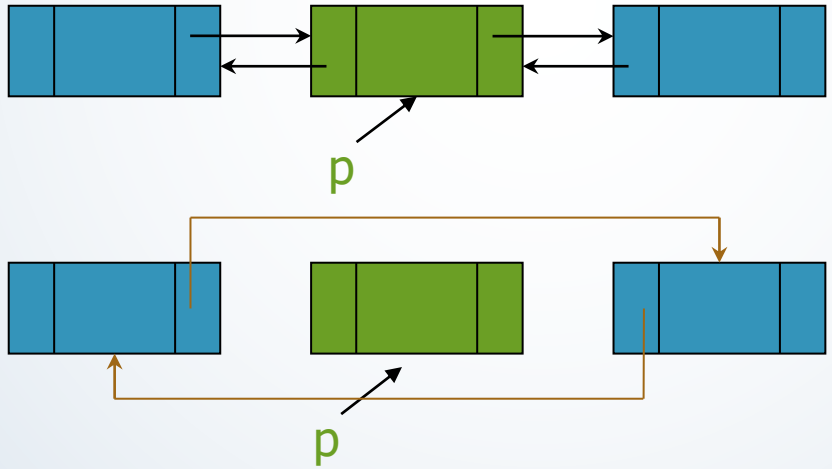
Doubly Linked List

- To insert a node after an existing node pointed by p



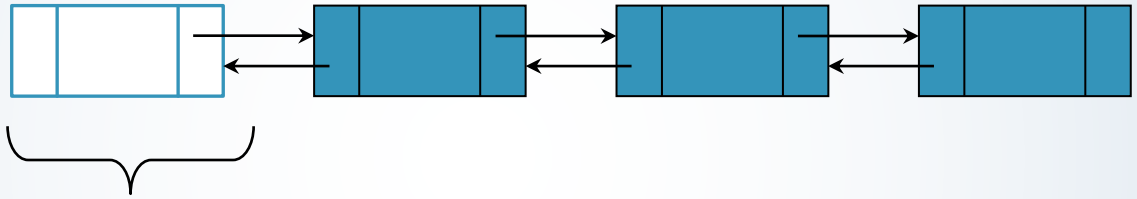
Doubly Linked List

- To delete a node, we only need to know a pointer pointing to the node



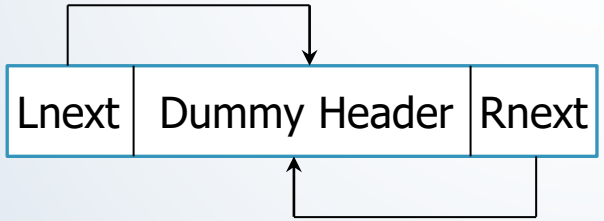
Further Extensions

- Doubly Linked List with Dummy Header

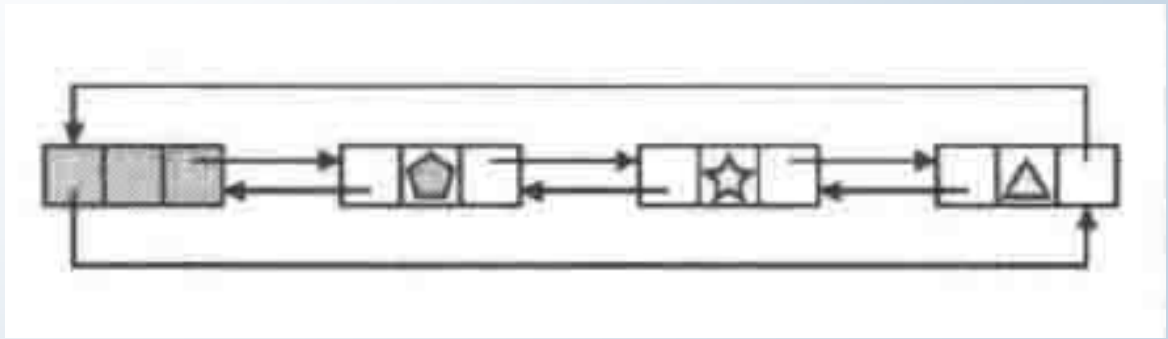


Dummy Header node

- When empty



- Doubly Circular Linked List?



Summary of Linked List

- **Linked allocation:**
  - Stores data as individual units and link them by pointers.
- **Disadvantages:**
  - Take up additional memory space for the links
  - Accessing random parts of the list is slow. (need to walk sequentially)

Advantages of linked allocation (1)

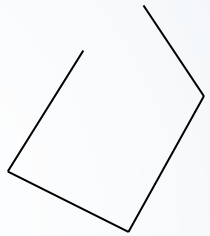
- Efficient use of memory
  - Facilitates data sharing
  - No need to pre-allocate a maximum size of required memory
  - No vacant space left
- Easy manipulation
  - To delete or insert an item
  - To join 2 lists together
  - To break one list into two lists

Advantages of linked allocation (2)

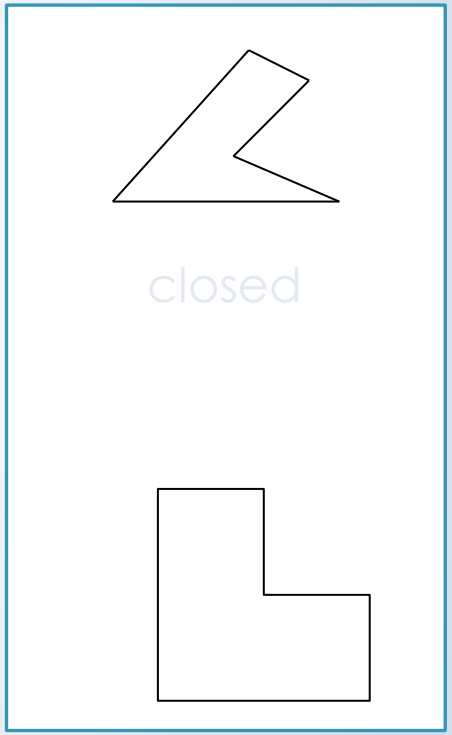
- Variations
  - Variable number of variable-size lists
  - Multi-dimensional lists (array of linked lists, linked list of linked lists, etc.)
- Simple sequential operations (e.g. searching, updating) are fast

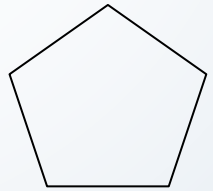
Applications: Representing Convex Polygons

- Polygon:
  - A closed plane figure with n sides.
- Convex:
  - A polygon is convex if it contains all the line segments connecting any pair of its points.



not closed

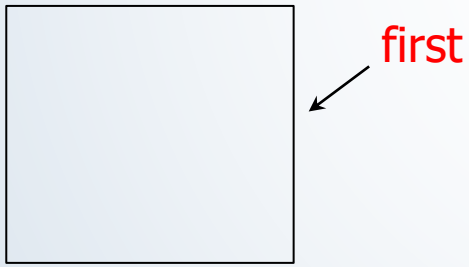


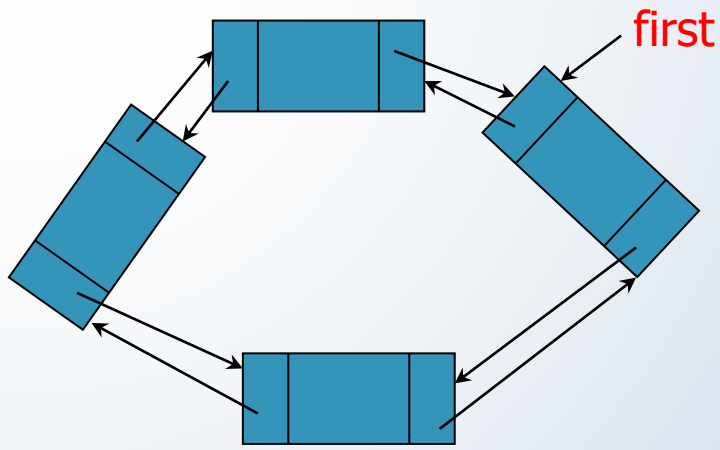


convex

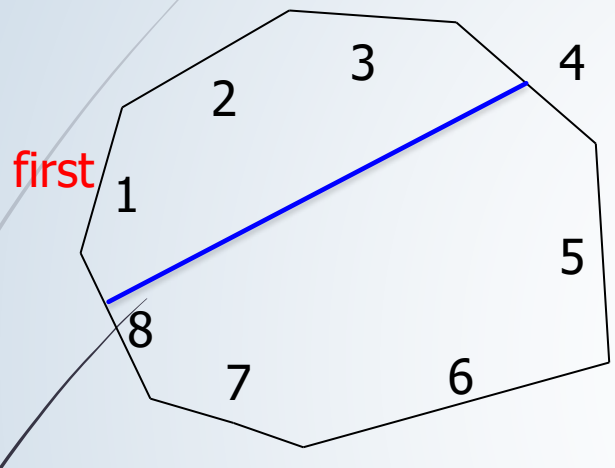
Convex Polygons

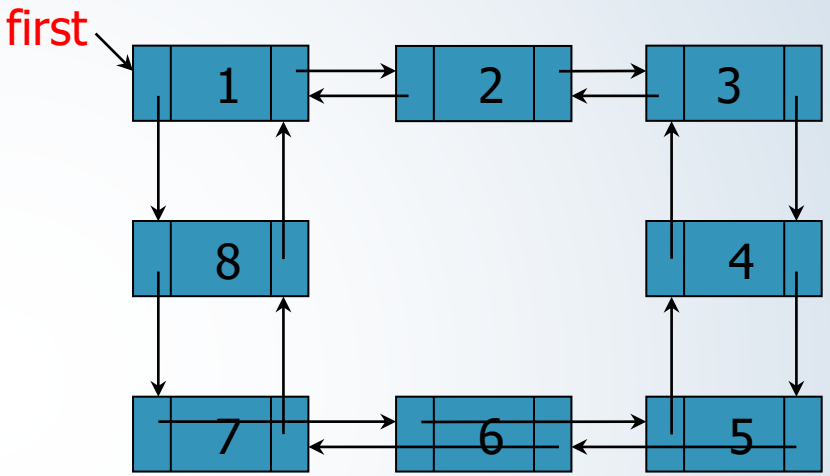
- Every doubly linked list node represents a line of the polygon.
- It is easy to handle partition.



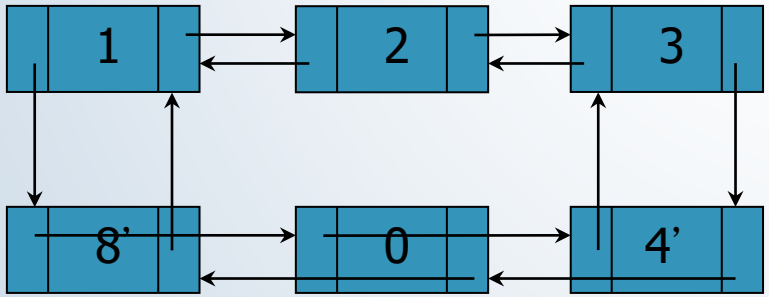


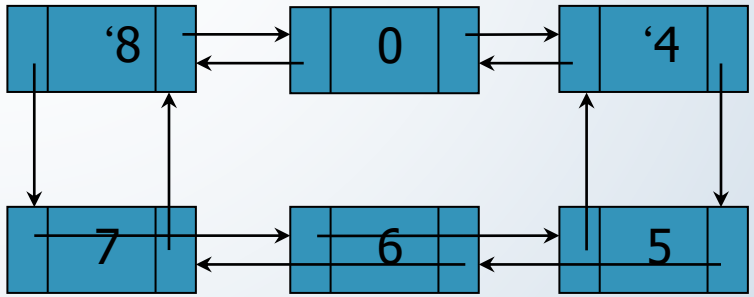
Cut a Polygon





After Cut





Node List ADT

- The Node List ADT models a sequence of positions storing arbitrary objects
- It establishes a before/after relation between positions
- Generic methods:
  - size(), empty()
- Iterator

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin	Return const_iterator to beginning (public member function)
cend	Return const_iterator to end (public member function)
crbegin	Return const_reverse_iterator to reverse beginning (public member function)
crend	Return const_reverse_iterator to reverse end (public member function)

Capacity:

empty	Test whether container is empty (public member function)
size	Return size (public member function)
max_size	Return maximum size (public member function)

Element access:

front	Access first element (public member function)
back	Access last element (public member function)

Modifiers:

assign	Assign new content to container (public member function)
emplace_front	Construct and insert element at beginning (public member function)
push_front	Insert element at beginning (public member function)
pop_front	Delete first element (public member function)
emplace_back	Construct and insert element at the end (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
emplace	Construct and insert element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
resize	Change size (public member function)
clear	Clear content (public member function)

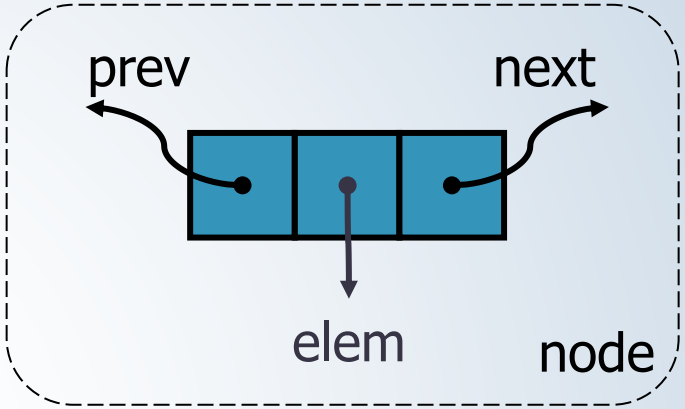
Operations:

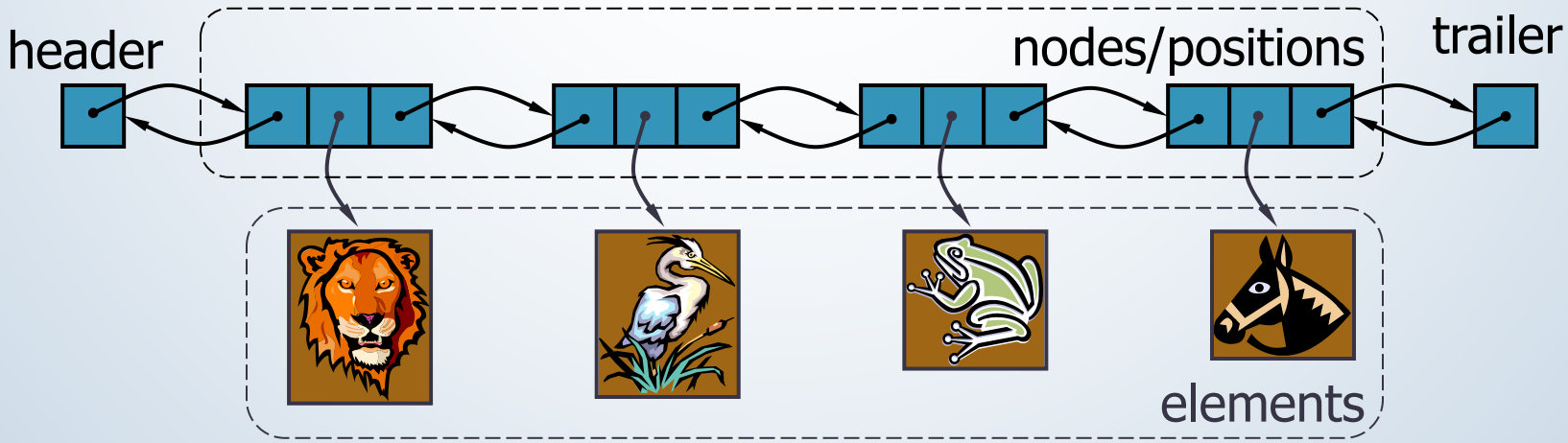
splice	Transfer elements from list to list (public member function)
remove	Remove elements with specific value (public member function)
remove_if	Remove elements fulfilling condition (public member function template)
unique	Remove duplicate values (public member function)
merge	Merge sorted lists (public member function)
sort	Sort elements in container (public member function)
reverse	Reverse the order of elements (public member function)

https://cplusplus.com/reference/list/list/

Implementation based on DLL

- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





Performance

- In the implementation of the List ADT by means of a doubly linked list
  - The space used by a list with n elements is $O(n)$
  - The space used by each position of the list is $O(1)$
  - The insert, erase operations of the List ADT run in $O(1)$ time

Lists in C++ STL

```
#include <list>
using std::list;                    // make list accessible
list<float> myList;                 // an empty list of floats
```

list(n): Construct a list with n elements; if no argument list is given, an empty list is created.

size(): Return the number of elements in L.

empty(): Return true if L is empty and false otherwise.

front(): Return a reference to the first element of L.

back(): Return a reference to the last element of L.

push_front(e): Insert a copy of e at the beginning of L.

push_back(e): Insert a copy of e at the end of L.

pop_front(): Remove the fist element of L.

pop_back(): Remove the last element of L.

Python Lists

- Python has a built-in list type, called “list”, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

- Like strings (and all other built-in sequence types), lists can be indexed and sliced.

```
>>> squares[0] # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:] # slicing returns a new list
[9, 16, 25]
```

Common List Methods

- list.append(elem) -- adds a single element to the end of the list
- list.insert(index, elem) -- inserts the element at the given index, shifting elements to the right.
- list.extend(list2) adds the elements in list2 to the end of the list. Using + or += on a list is similar to using extend().
- list.index(elem) -- searches for the given element from the start of the list and returns its index.
- list.remove(elem) -- searches for the first instance of the given element and removes it
- list.sort() -- sorts the list in place (does not return it).
- list.reverse() -- reverses the list in place (does not return it)
- list.pop(index) -- removes and returns the element at the given index. Returns the rightmost element if index is omitted (roughly the opposite of append()).

Python Lists

- A Python list is built as an array. Even though you can do many operations on a Python list with just one line of code, there's a lot of code built in to the Python language running to make that operation possible.
- For example, inserting an item at the front of the list.
- list.insert(index, elem)

inserts the element at the given index, shifting elements to the right.

what is the time complexity of insert( )?

- Time complexity of various operations in Python can be found here.