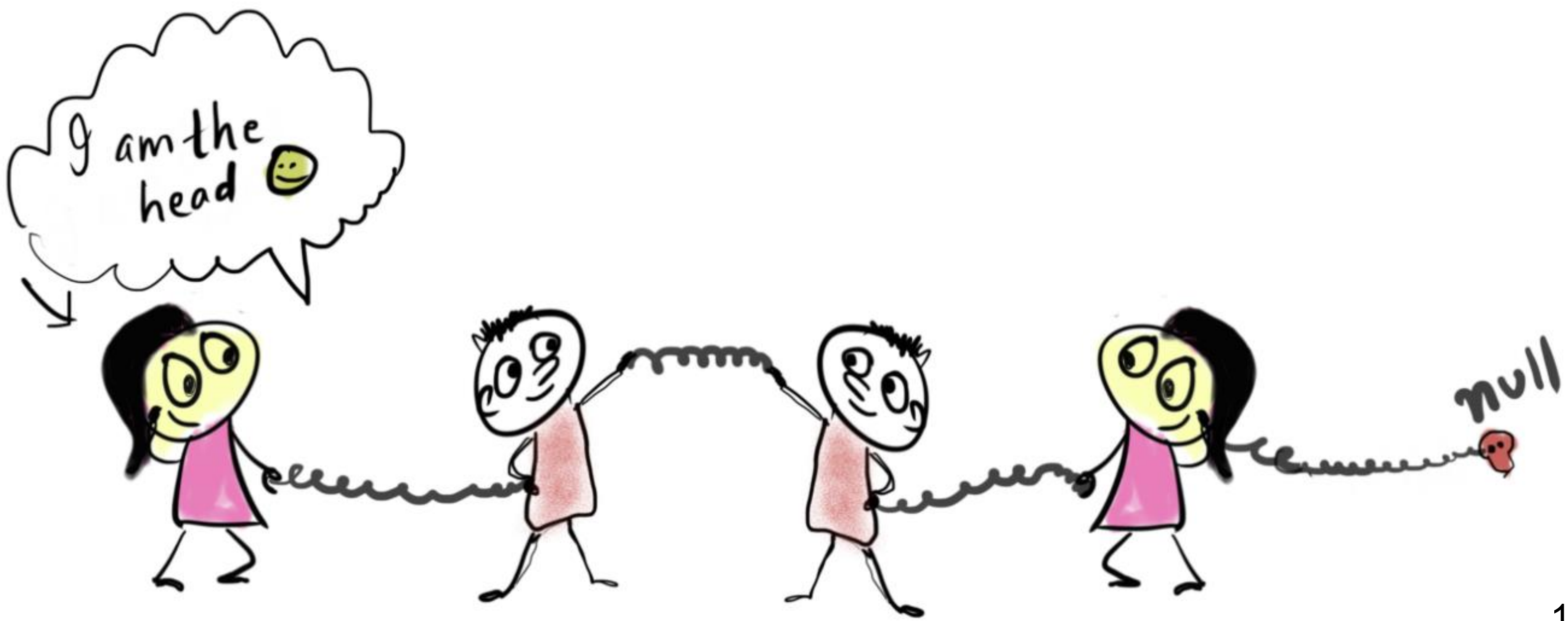
EE2331 Data Structures and Algorithms

Linked List, Stack, Queue: non-random access linear data structure

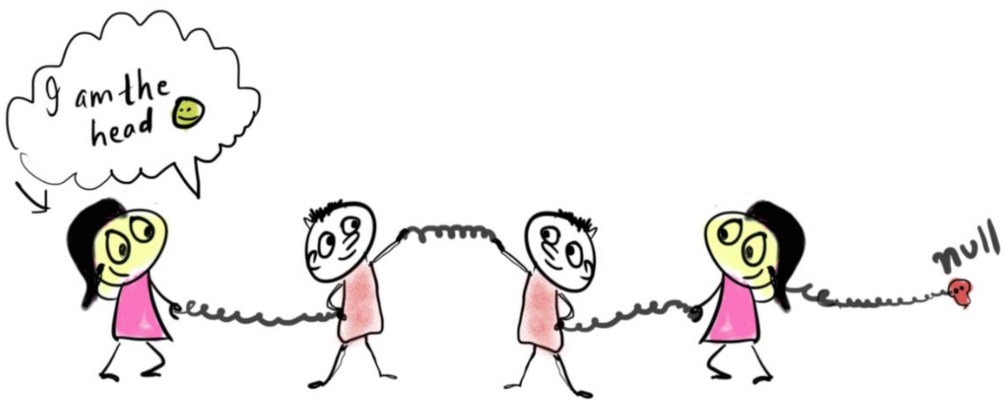


Compared to array:

No random access

- Array A[5]={1,2,3,4,0}
  - A[0]=1, A[2]=3, etc.

- Linked list, stack, queue
  - Lack direct access to the elements by their positions
  - has to iterate from a known position (like the beginning or the end) to that position, which takes linear time in the distance between these



Linear List

- Each element in the list has a unique predecessor (previous) and successor (next).
- Unordered/Random list
  - There is no ordering of the data.
- Ordered list
  - The data are arranged according to a key. A key is one or more fields within a structure that is used to identify the data or otherwise control its use.
- General list
  - Data can be inserted and deleted anywhere and there are no restrictions on the operations that can be used to process the list.
- Restricted list
  - Insertion, deletion and processing of data are restricted to specific locations, e.g. the two ends of the list. Stack and Queue are examples of restricted list.

Josephus Problem

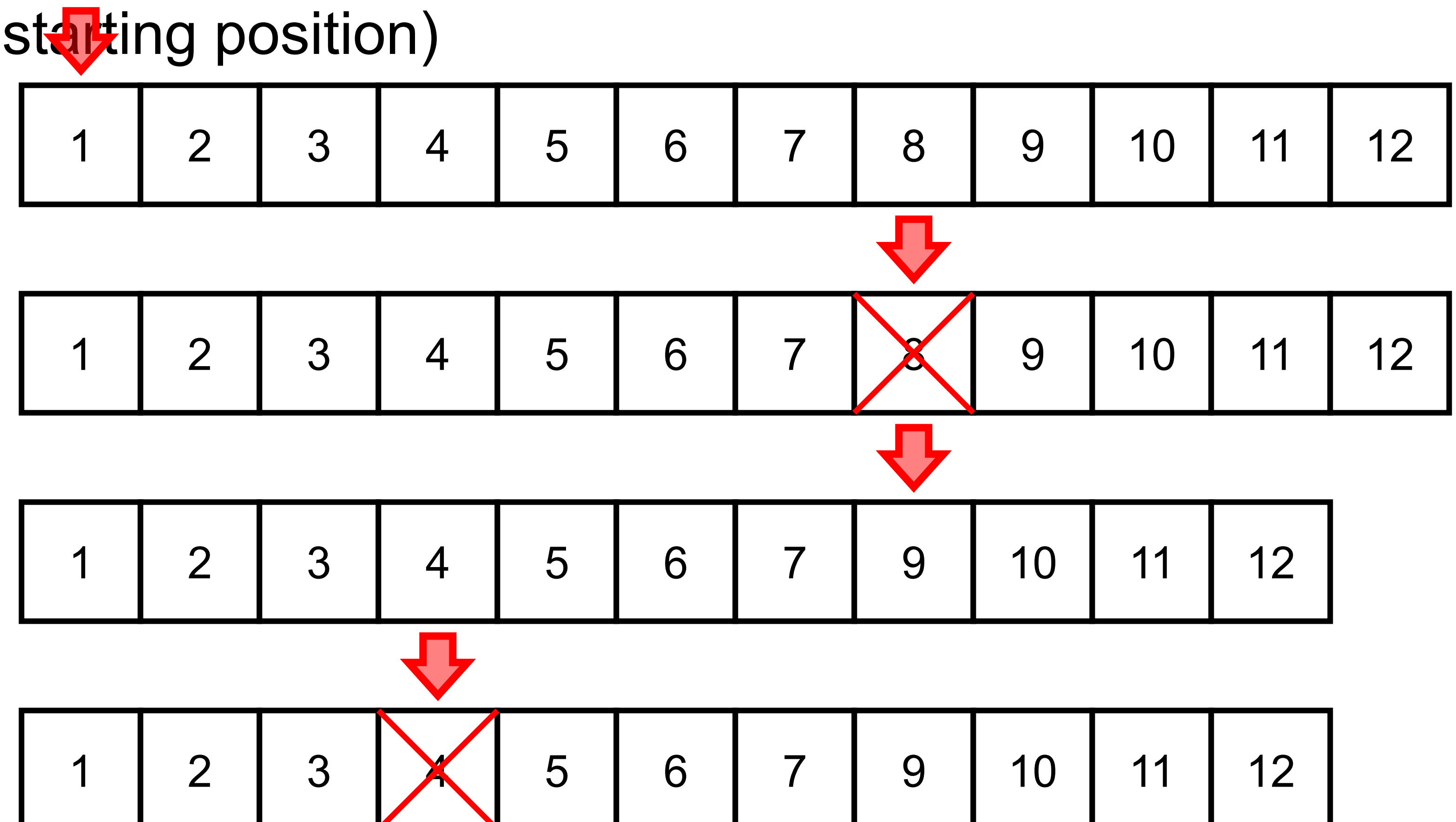
- People are standing in a circle waiting to be executed. Counting begins at a specified point in the circle and proceeds around the circle in a specified direction. After a specified number of people are skipped, the next person is **executed**. The procedure is repeated with the remaining people, starting with the next person, going in the same direction and skipping the same number of people, until only one person remains, and is **freed**.
- The problem — given the number of people (***n***), starting point, direction, and number to be skipped (***k***) — is to choose the position in the initial circle to avoid execution (i.e. **guessing who is the survivor**).

History (from wiki)

- The problem is named after Flavius Josephus, a Jewish historian living in the 1st century. According to Josephus' account of the siege of Yodfat, he and his 40 soldiers were trapped in a cave by Roman soldiers. They chose suicide over capture, and settled on a serial method of committing suicide by drawing lots.

The Josephus Problem

- If k = 7, n = 12 (skip 7 positions including the starting position)



Array Implementation

- A simple approach is by writing a program to simulate the counting-out game. But what data structure should be used?
- With a list using array implementation
  - Array has the advantage of random access (i.e. direct access to any position)
  - However, the insert and delete operation may involve substantial data movement
  - Another disadvantage of representing a list using an array is that the maximum length of the list needs to be determined a priori

Linked List & Node Structure

- A sequence of nodes (elements), each containing arbitrary data and links (pointers) pointing to the next and/or previous nodes

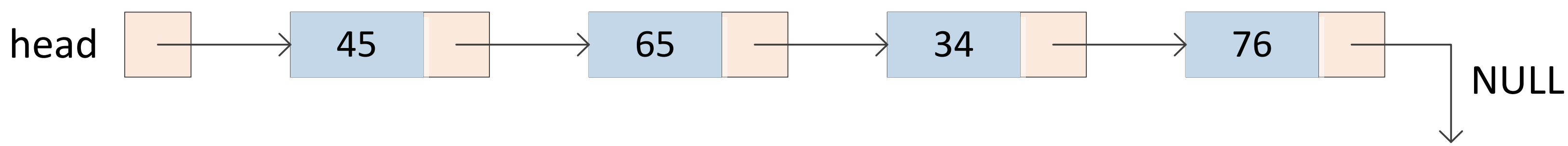
data	link

- It contains pointer(s) of the same type
  - Recursive data structure (self-referential datatype)
- A list is formed by linking nodes together in a sequential manner
- In typical C++ implementations, we shall define the node using struct, and the linked list is defined as a class.

```
struct node {
   int info;         // data
   node* link;
};

node* head;          // head pointer pointing to first node
```

Linked List Example

- In the C++ terminology, a linked list is classified as a container.
- The address of the first node in the list is stored in a separate pointer variable usually called *head*, *first*, or *list*.
- The null pointer (NULL, physical value 0 in C/C++) is used to denote the end of the list, or not a valid address.
- Example: a linked list of 4 integers.



Common Operations on Linked List

https://yongdanielliang.github.io/animation/web/LinkedList.html

Find Length & Find Node

- To find the length of the linked list

```
int len = 0;
node *cur = head;          //traverse the list using cur

while (cur != NULL) {      // cur points to a valid node
   len++;
   cur = cur->link;        //move to the next node
}
```

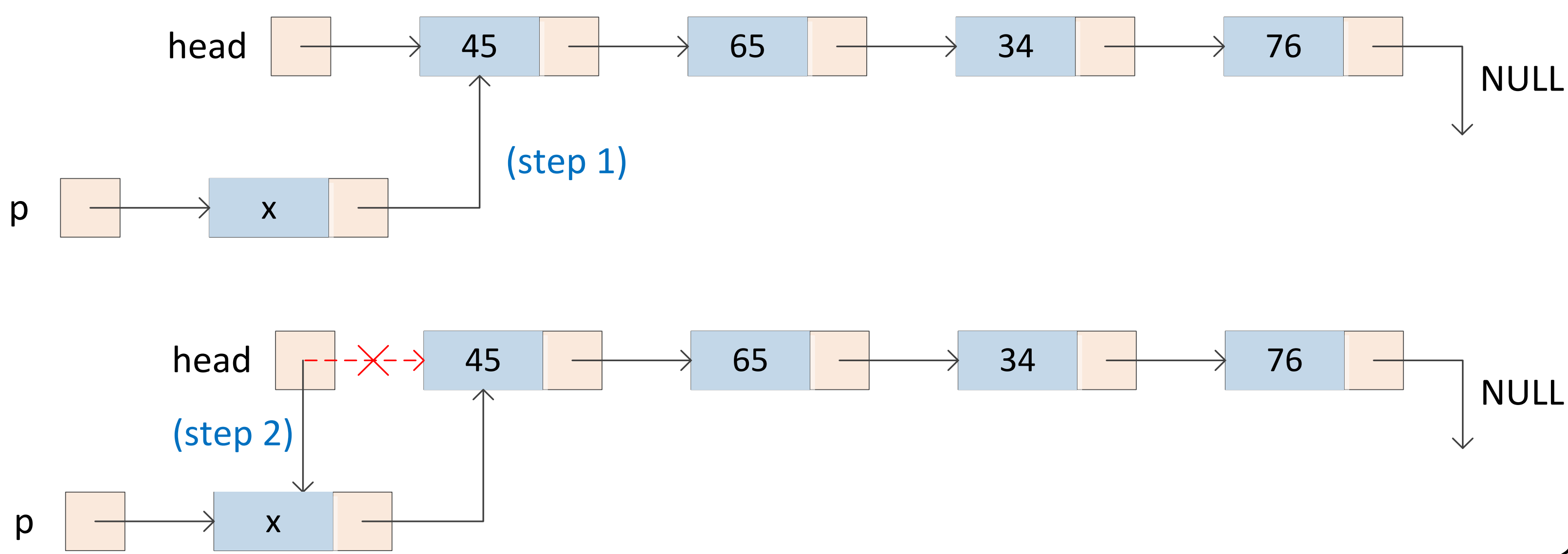
- To search an element x from the beginning of the list

```
node *cur = head;

while (cur != NULL && cur->info != x) {          // search x
   cur = cur->link;
}
// cur now points to target x or is NULL (x not exist)
```

Insert Node (at front)

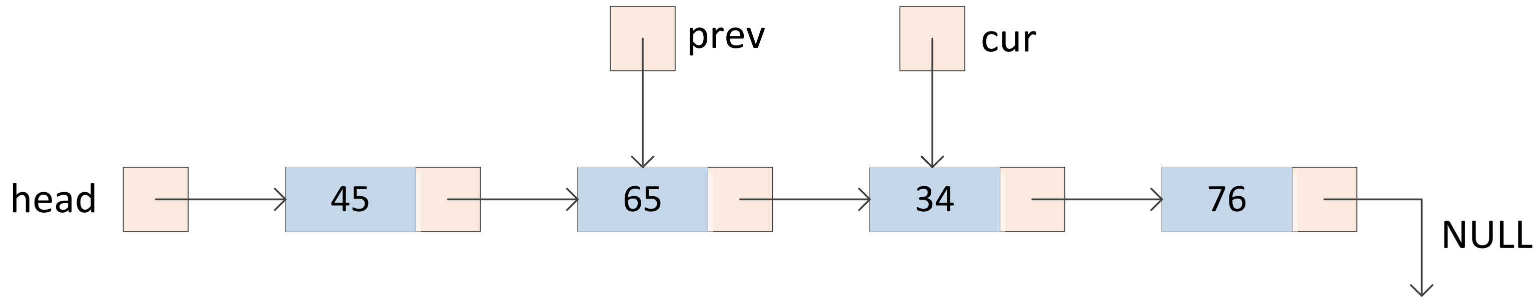
- Insert a new element x at the front of the list

```
node *p = new node;      // create the node dynamically for storing x
p->info = x;

p->link = head;          // step 1
head = p;                // step 2
```

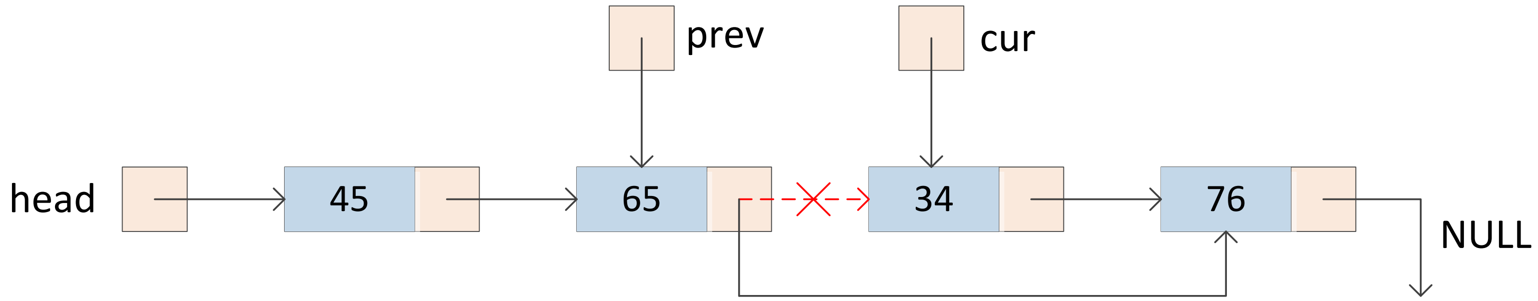


Remove Node

1. Locate the node storing the value x, e.g. x = 34

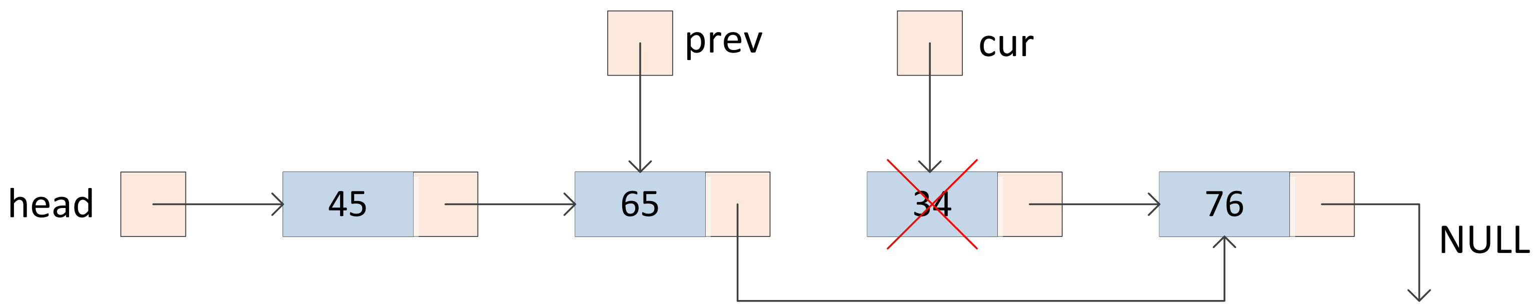


2. Update the links

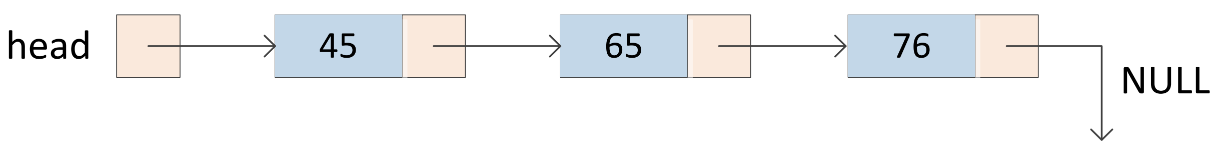


Remove Node (cont.)

3. Physically delete the node



4. Structure of the list after removing the element 34



Remove Node (cont.)

- Remove the element x (1st instance) from the linked list
- To remove a node from the linked list, we need to know the reference to its predecessor.

```
node *cur = head;
node *prev = NULL;          // prev points to the predecessor of cur

while (cur != NULL && cur->info != x) {                // search x
   prev = cur;
   cur = cur->link;
}

// if cur == NULL, x is not found in the linked list
if (cur != NULL) {          // cur->info == x
   if (prev != NULL)        // why checking this?
      prev->link = cur->link;         // skip cur node
   else                     // cur is the first node in the list
      head = cur->link;     // x is the first node
   delete cur;              //free the storage of the removed node
}
```

Insert Node

- Insert a new element x into an ordered list

```
node *p = new node;
p->info = x;

if (head == NULL || x <= head->info) {
   p->link = head;           //insert at front
   head = p;
} else {                     //head != NULL && x > head->info
   node *prev = head;        // x to be inserted between prev and cur
   node *cur = head->link;   // i.e. prev->info < x <= cur->info

   while (cur != NULL && x > cur->info) {     // search position
      prev = cur;
      cur = cur->link;
   }

   // end-of-list OR x <= cur->info, so insert node p after node prev
   p->link = prev->link;
   prev->link = p;
}
```

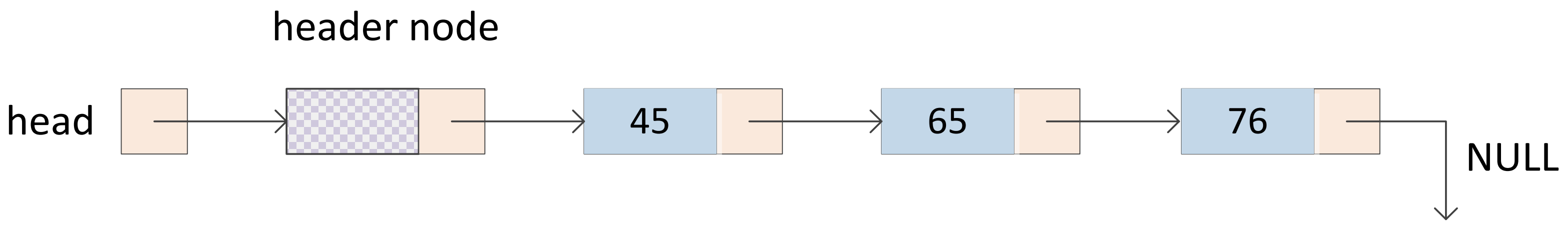
Remark

- Because you can't go backward in a singly linked list, for removing/inserting node, you usually need to use a pair of pointers – *predecessor* and *current*, to keep track of the previous node and update its link.
- For a linked list of size n, you generally should test your algorithm against the two boundary cases: n=0 and n=1 in addition to the general case.
- **Common Problems**
  - Null-pointer exception is a common error in programs that manipulate linked list.
  - A pointer must be properly initialized or tested for not equal to NULL before you can use it to access a data member (dereferencing) or the next node.
  - Broken list due to deletion of nodes
  - Losing reference to some nodes (memory leak)

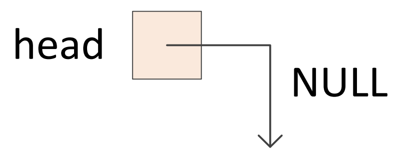
Other Variants of Linked List

Linked List with Header Node

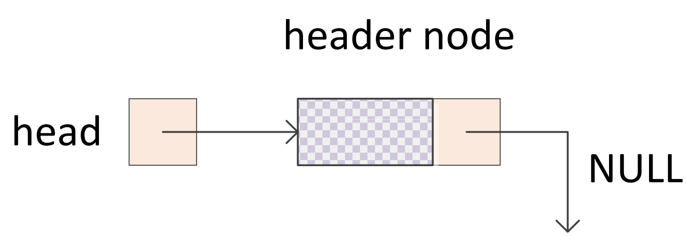
- The data field of the header node is NOT used to store valid data. Some metadata may be stored in the header node.



- List does not exist (or not yet created):



- List is empty:



Why Header Node?

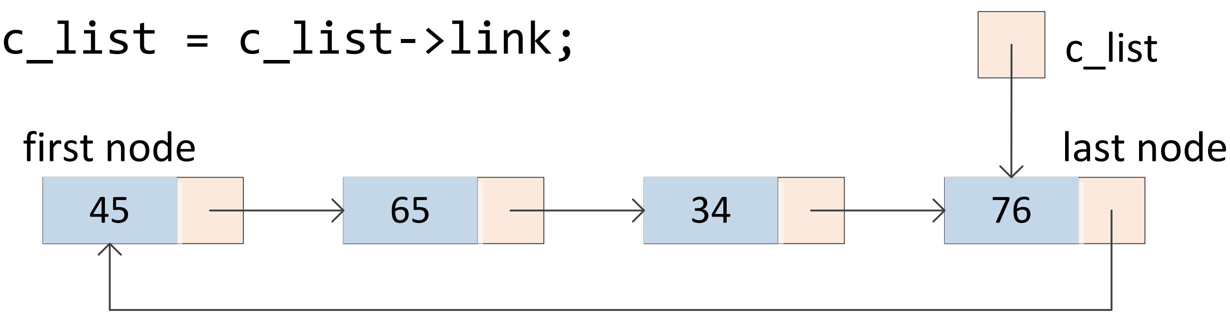
- Header node is guaranteed to exist at all times.
- Header node makes all list nodes intrinsically the same – having a predecessor.
- Having a sentinel zeroth node simplifies a lot of operations you might want to perform on a linked list - for example, a lot of operations no longer need to explicitly check for an empty list.

```
// Simplified version: Insert a new element x into an ordered list
node *p = new node;
p->info = x;

node *prev = head;          // point to the dummy header
node *cur = head->link;     // point to 1st node

while (cur != NULL && x > cur->info) {        // search position
   prev = cur;
   cur = cur->link;
}
p->link = prev->link;
prev->link = p;
```

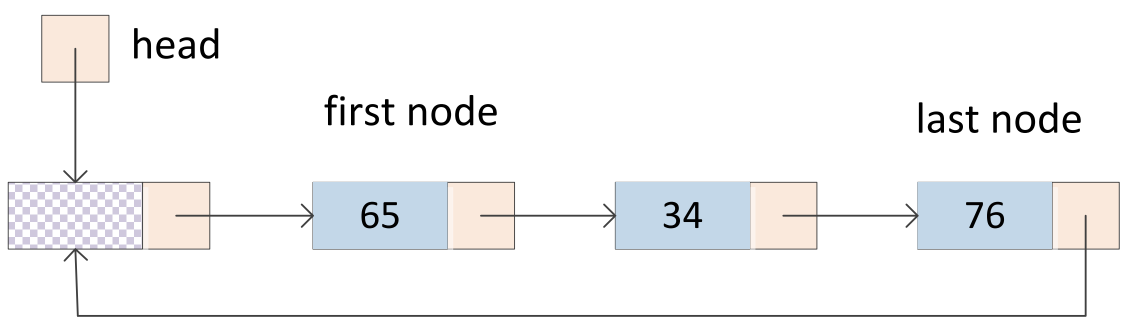
Circularly Linked List (not required)

- The link of the last node points back to the first node.
- When the pointer ***c_list*** reaches the last node in the list, it can re-visit the first node in one step:

```
c_list = c_list->link;
```



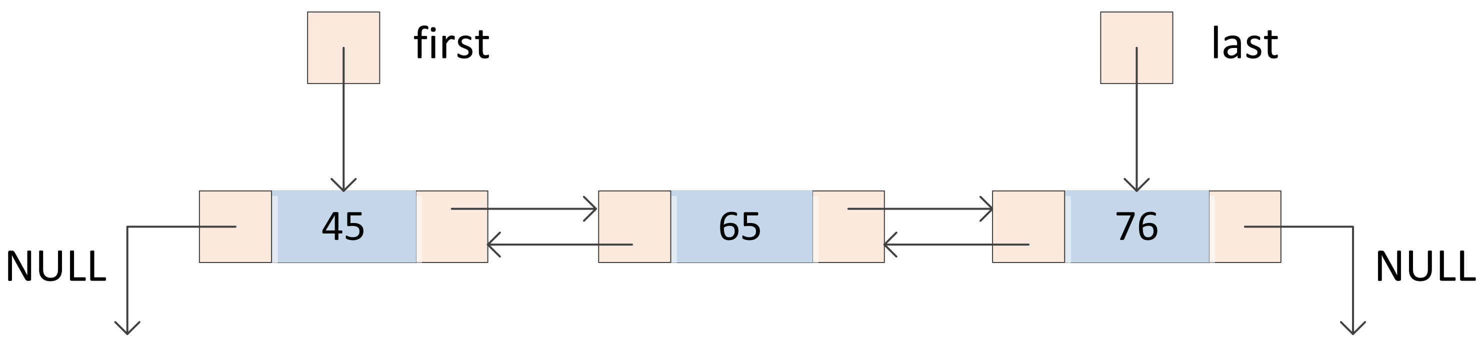
- Circular list with header



Doubly Linked List

- With a doubly-linked list, we can traverse the list in the forward or backward direction.

```
template<class Type>
struct nodeType {
   Type info;
   nodeType<Type> *next;          //points to successor node
   nodeType<Type> *back;          //points to predecessor node
}
```



Insert Node on Doubly Linked List (not required)

- Insert element x after the current node in a doubly-linked list

```
nodeType<Type> *p = new nodeType<Type>;
p->info = x;

// assume current is a pointer to current node
p->next = current->next;     //(1)
p->back = current;           //(2)

if (current->next != NULL)   //current has a successor
   current->next->back = p;  //(3)

current->next = p;           //(4)
```

Remove Node on Doubly Linked List (not required)

- Remove the node pointed by ***p*** in a doubly-linked list

```
if (p->next != NULL)
   p->next->back = p->back;          //(1)

if (p->back != NULL)
   p->back->next = p->next;          //(2)

delete p;
```

In-Class Exercise

- Return the value/info of the last node of a list
- Remove last node of a list

- The given singly linked list has a ***dummy header***
  - Input: a pointer to node structure, *list*, which points to the head of a linked list
  - Precondition: *list* is a valid linked list with header

Search the Last Node

```
// Output: a pointer p points to the last node of the list
node* searchLastNode(node *list) {

    node *p;
    p = list->link;

    if(p == NULL)                        // empty list
        return p;

    while (p->link != NULL)              // reach the end
        p = p->link;

    return p;



}
```

Remove the Last Node

```
// singly linked list with header hode
void removeLastNode(node *list) {

}
```

Linked List C++ Implementation

linkedListType.h
in Files

- Operations that we would perform on a linked list:
- Initialize the list.
- Clear the list.
- Determine if the list is empty.
- Print the list.
- Find the length of the list.
- Make a copy of the list, e.g. assignment operator= and the copy constructor.
- Search the list for a given item.
- Insert an item to the list.
  - The requirement of the insert operation depends on the representation invariant or the intended uses of the list.
  - For ordered list, we need to maintain the ordering of list elements.
  - If it is used as a queue, insertion is performed at the rear (end of list).
  - If it is used as a stack, insertion is performed at the front.

Linked List C++ Implementation

- Remove an item from the list. Similar to the case of insertion.
- Traverse the list (in the application program that uses the linked list object), i.e. retrieve the elements one by one (in some specific order) to carry out the required computation on each node.
  - To implement the traversal, we shall make use of an iterator.
  - A linked list is a container that holds together a collection of items.
  - An iterator is an object that produces each element of a container, one at a time.
  - The two basic operations on an iterator are the dereference operator *, and the pre-increment operator ++ (advance to the next element).
- There can be other operations on the linked list, e.g. reverse the list, merge two lists, etc.

- We want the linked list class to be generic such that it can be used to process different data types.

About using reference for a pointer

```
/* use reference when you need to change the passed parameters for a function

   function(type1& parameter1, type2 parameter2)
   {
       parameter1=.... //modify parameter1
   }

   main....
   {
     type1 x1;
     type2 x2;
     function(x1, x2);

     //x1 is modified accordingly

   }
 */
```

Reference of pointer: Example
(refer to the sample code for tutorial week 3)

```cpp
void insert(ListNode*& head, int x) {
        ListNode* p = new ListNode;
        p->info=x;
        p->link=NULL;

        if (head == NULL || x <= head->info) {
                p->link = head;
                head = p;
        }
        else {
                ListNode* prev = head;
                ListNode* cur = head->link;

                while (cur != NULL && x > cur->info) {
                        prev = cur;
                        cur = cur->link;
                }

                p->link = prev->link;
                prev->link = p;
        }
}
```

```cpp
int main() {
  ListNode* head = NULL;
  ifstream inFile("testData.txt");

  if (!inFile.is_open()) {
    cout << "Error: cannot open data file" << endl;
    exit(0); //terminate the program
  }
  while (!inFile.eof()) { //not end of file
    int i;
    inFile >> i;    //read in an integer

    if (!inFile.fail())
      insert(head, i); //insert into the linked list
    else
      break;
  }
  inFile.close();
}
```

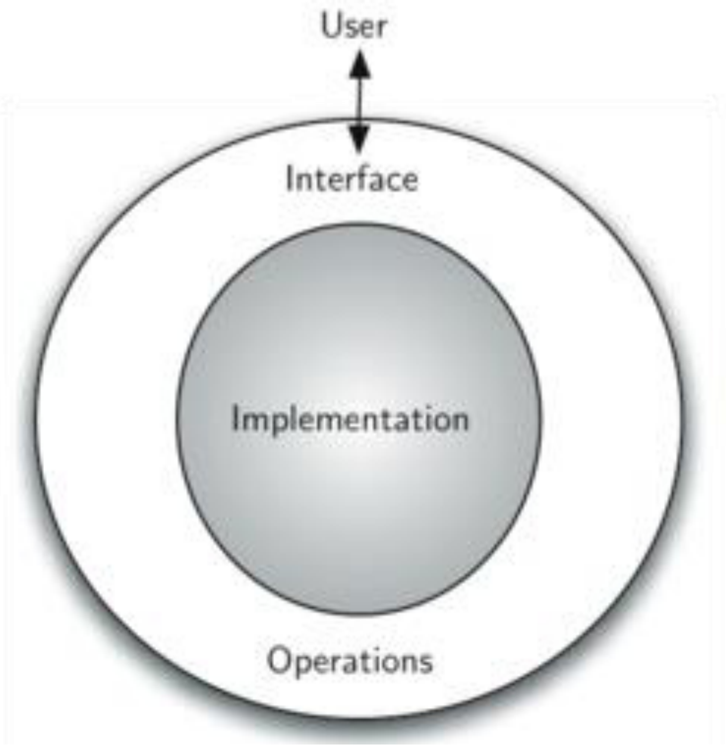
Use linked list in C++ STL

Abstract Data Type (ADT)

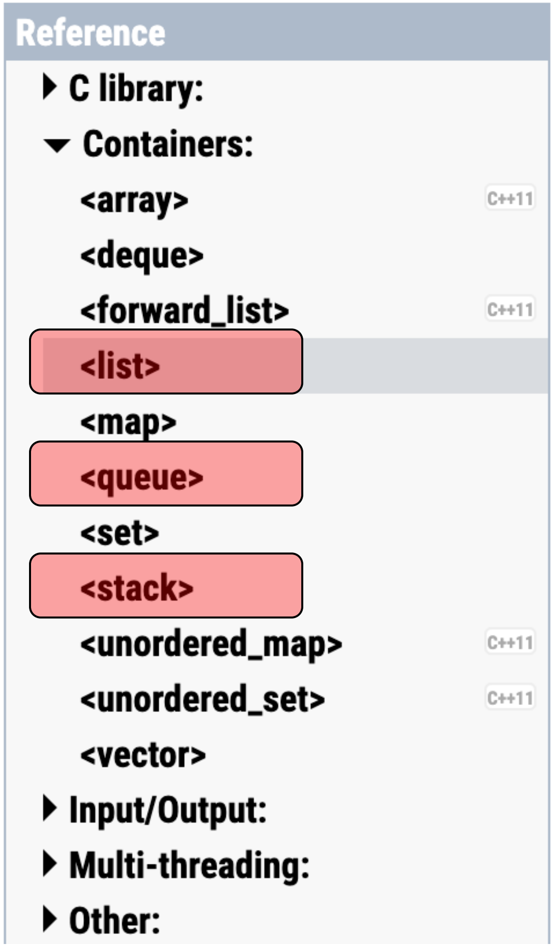
- To manage the complexity of problems and the problem-solving process, computer scientists use abstractions to allow them to focus on the “big picture” without getting lost in the details.
- **Abstract Data Type** is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. This means that we are concerned only with what the data is representing and not with how it will eventually be constructed.
- For example, the standardized user interface of an Android phone is a logical property of the device, while the construction of the physical Android phone is the implementation details. From the point of view of the user, you only need to know the logical property (i.e. the user interface) of the device when you are using the phone, and you don’t need to know its internal implementation details.

Abstract Data Type (ADT)

- This provides an **implementation-independent** view of the data. Since there will usually be many different ways to implement an abstract data type, this implementation independence allows the programmer to switch the details of the implementation without changing the way the user of the data interacts with it. The user can remain focused on the problem-solving process.



https://cplusplus.com/reference/stl/



List containers are implemented as doubly-linked lists

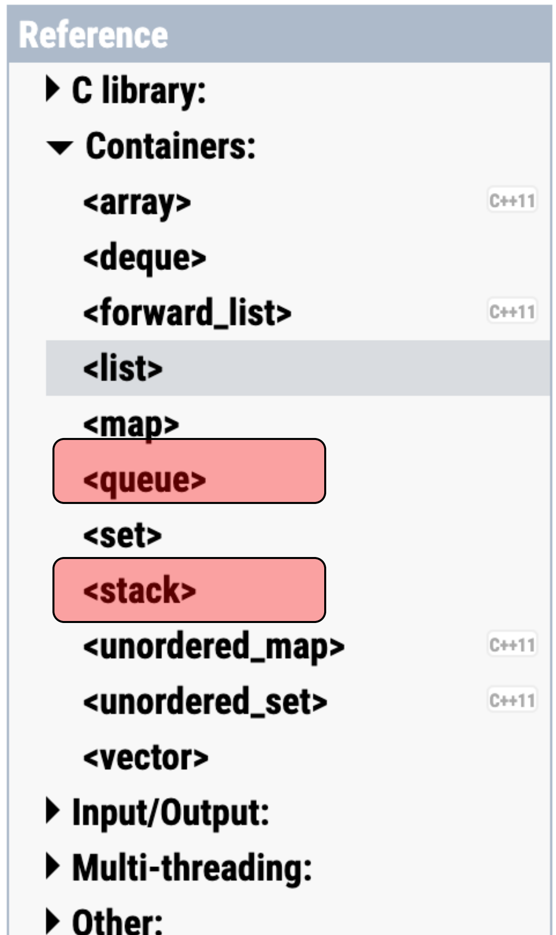
Compared to other base standard sequence containers (array, vector), lists perform generally better in inserting, extracting and moving elements in any position within the container for which an iterator has already been obtained, and therefore also in algorithms that make intensive use of these, like sorting algorithms.
they lack direct access to the elements by their position; For example, to access the sixth element in a list, one has to iterate from a known position (like the beginning or the end) to that position, which takes linear time in the distance between these. They also consume some extra memory to keep the linking information associated to each element (which may be an important factor for large lists of small-sized elements).

Example

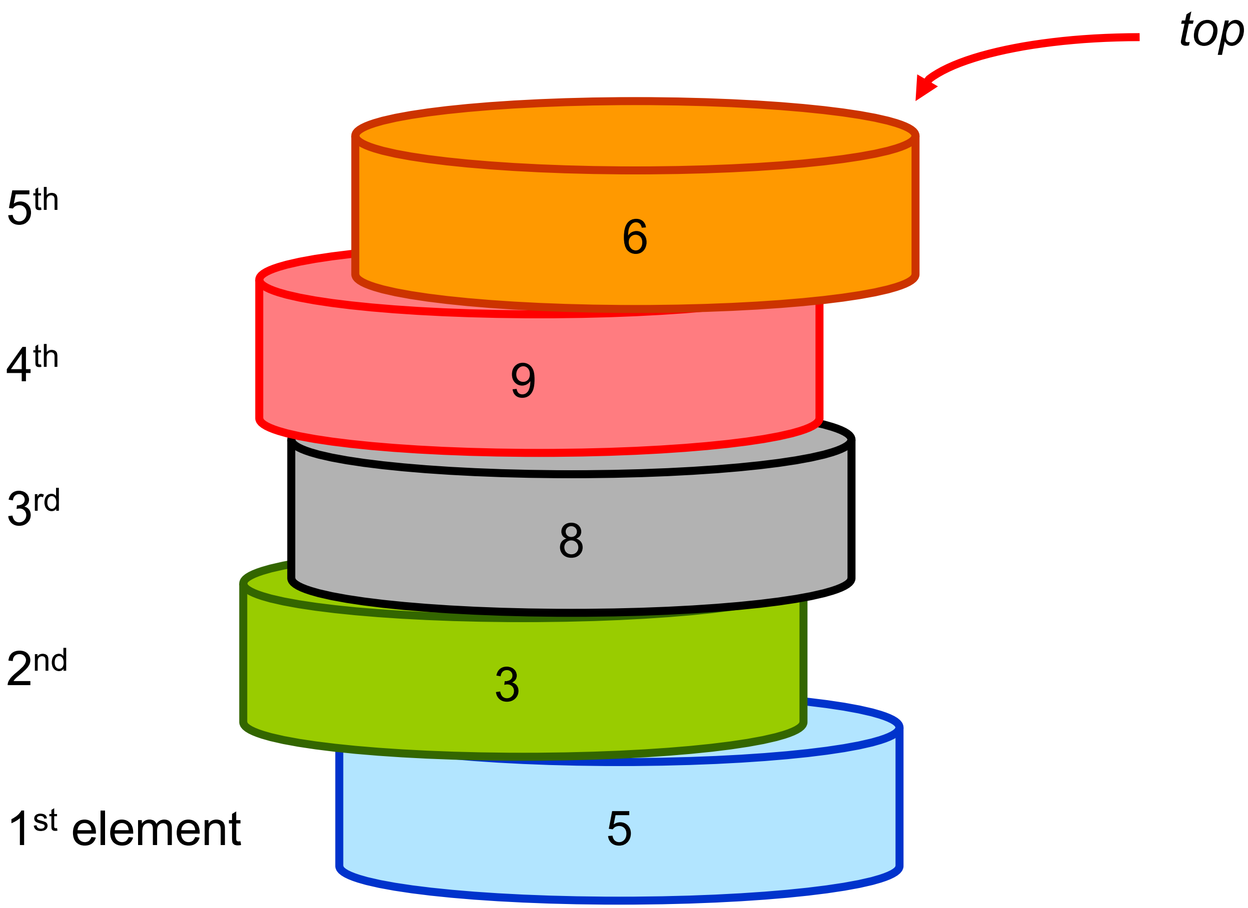
```
// constructing lists
#include <iostream>
#include <list>

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

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

  std::cout << '\n';

  return 0;
}
```

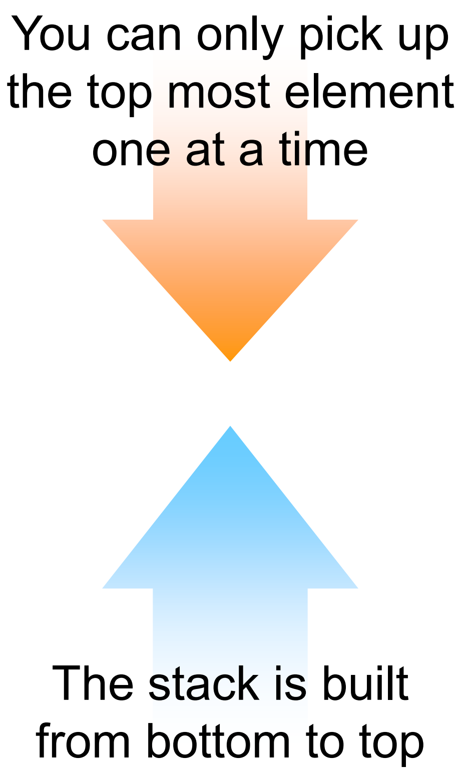
Stack and queue

https://cplusplus.com/reference/stl/



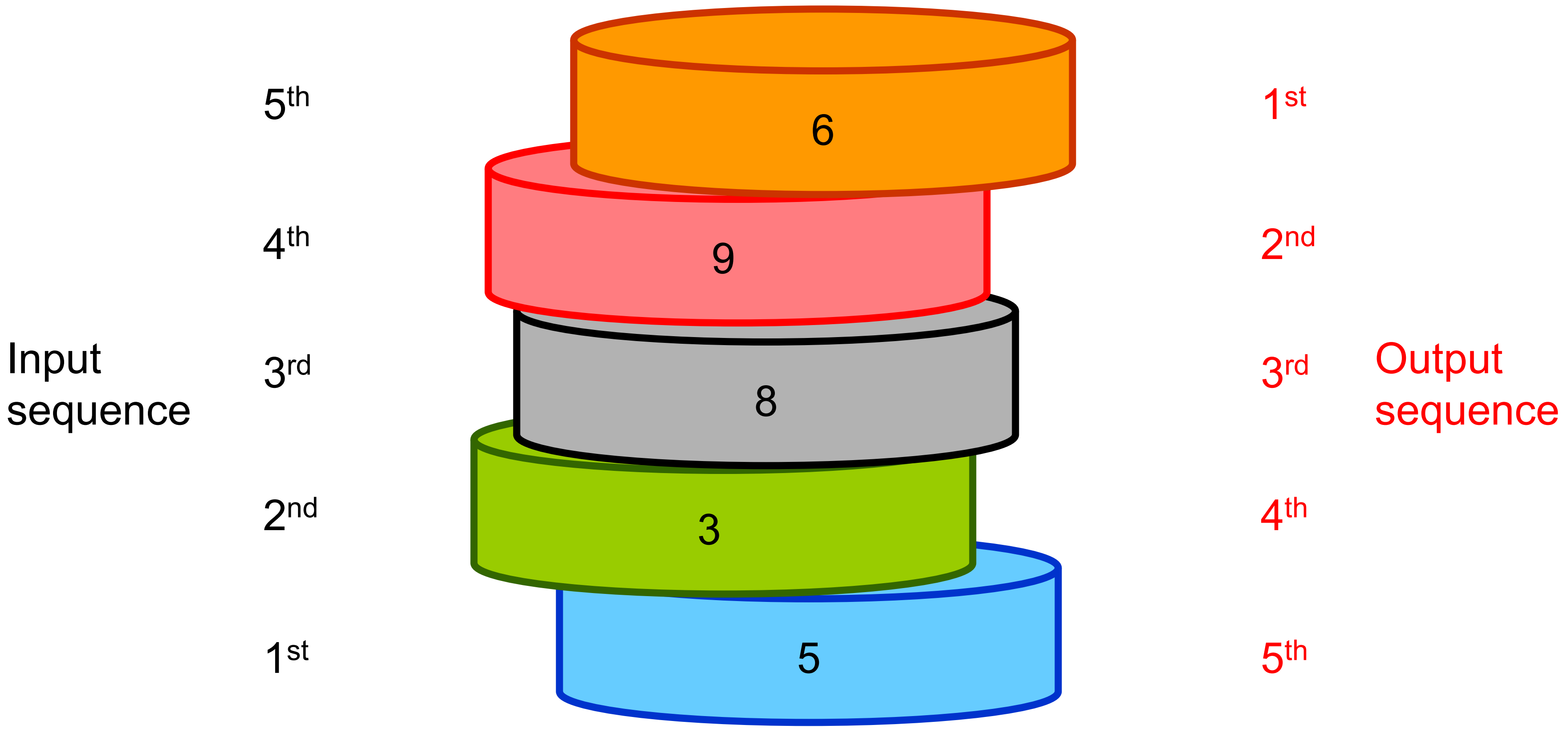
Stack


top
5th
6
4th
9
3rd
8
2nd
3
1st element
5


You can only pick up the top most element one at a time
The stack is built from bottom to top

Input/Output Order

- Last In First Out (LIFO)



Stack Operations

- A stack is a list of homogeneous elements in which the addition and deletion of elements occur only at one end, called the top of the stack.
- A stack is also called a Last In First Out (LIFO) data structure.
- Operations on a stack:
  - **initialize**: initialize the stack to an empty state
  - **size**: determine the number of elements in the stack
  - **empty**: determine if the stack is empty
  - **top**: retrieve the value of the top element
  - **push**: insert element at the top of stack
  - **pop**: remove top element
- In C++, we can define an ADT using an abstract class. In our discussion, I will try to follow the notations used in the C++ STL (Standard Template Library).

Using Stack to Reverse Order

- Use the class ***stack*** in C++ STL

```
#include <iostream>
#include <stack>

using namespace std;

int main() {
        stack<int> s;
        s.push(10);
        s.push(20);
        s.push(30);

        while(!s.empty()) {
                cout << s.top() << " ";     // output: 30 20 10
                s.pop();                    // remove the top item
        }
}
```

Using Stack to Evaluate Arithmetic Expression

- How does a computer evaluate this?
  - (4 + 5) * (7 - 2)
- In infix format, the binary operator is placed in between the 2 operands. The order of evaluation is determined by the precedence relation of the operators and parentheses, if any.
  - Order of precedence: ( ) > *, / > +, -
- Postfix notation is another way of writing arithmetic expressions, where the operator is written after the two operands:
  - e.g. 4 + 5 (infix) will be changed to 4 5 + (postfix)
  - The order of evaluation is the same as the order in which the operators appear in the postfix expression.
  - Precedence rules and parentheses are never needed!

Evaluate Postfix Expressions

- In the examples shown below, $ represents the exponentiation operator.

Infix	Postfix
$A + B$	$AB +$
$A + B - C$	$AB + C -$
$(A + B) * (C - D)$	$AB + CD - *$
$A \$ B * C - D + E / F / (G + H)$	$AB \$ C * D - EF / GH + / +$
$((A + B) * C - (D - E)) \$ (F + G)$	$AB + C * DE - - FG + \$$
$A - B/(C * D \$ E)$	$ABCDE \$ * / -$

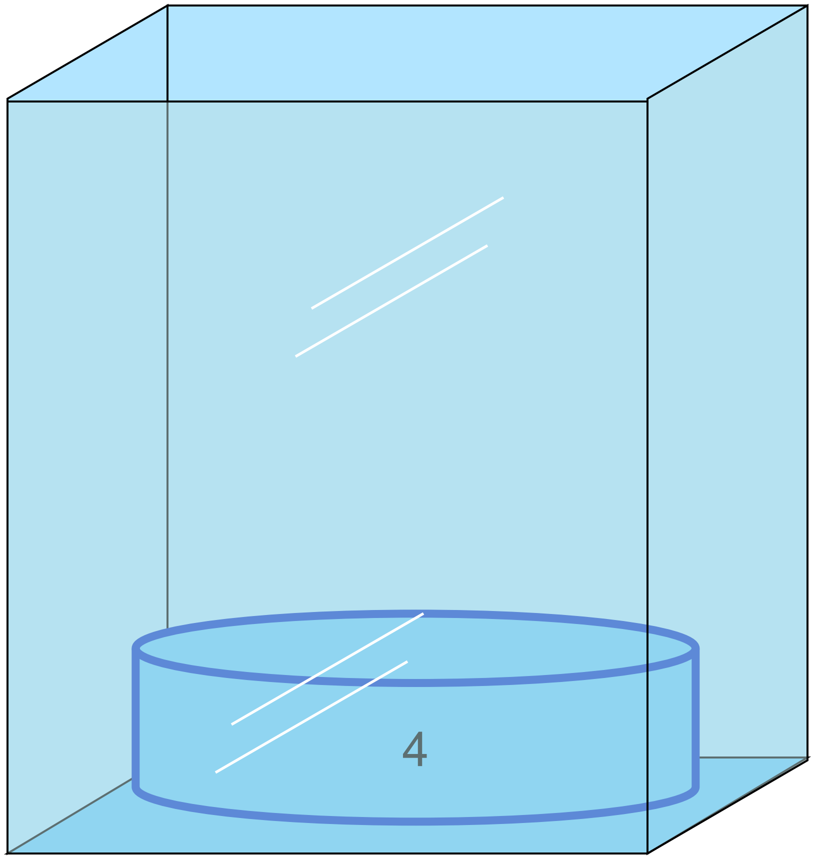
- Read from postfix
  - If input is an operand, push on stack
  - If input is an arithmetic operator
    - pop from stack twice (the two nearest operands)
    - compute their result
    - push the result onto stack

Example

Infix: (4 + 5) * (7 - 2)

Postfix: 4 5 + 7 2 - *

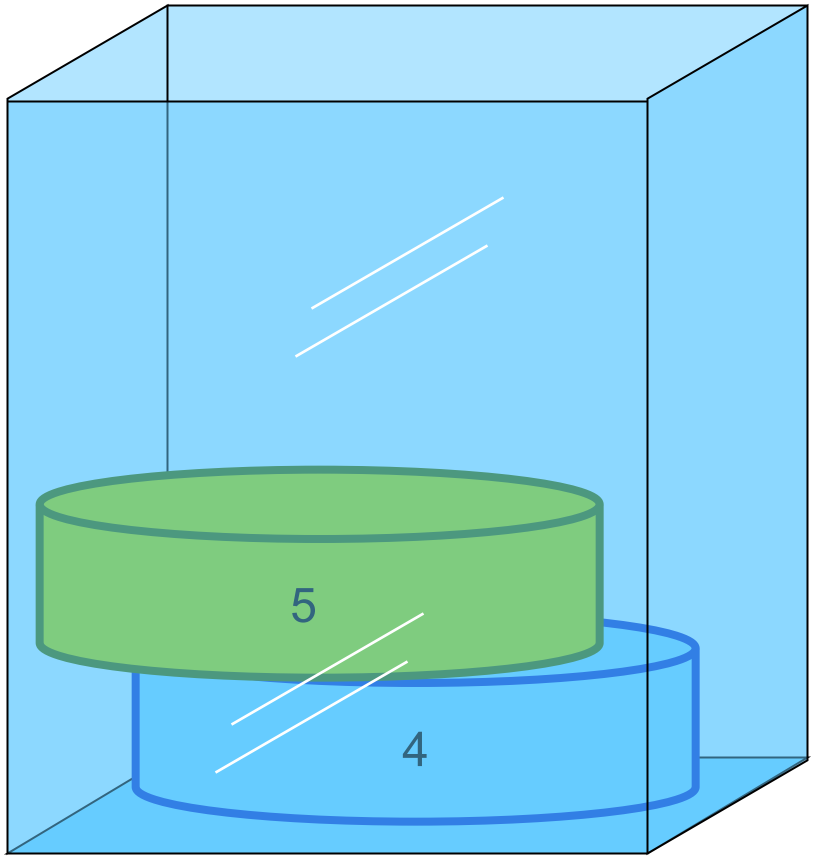
4 5 + 7 2 - *



Step 1: push(4)

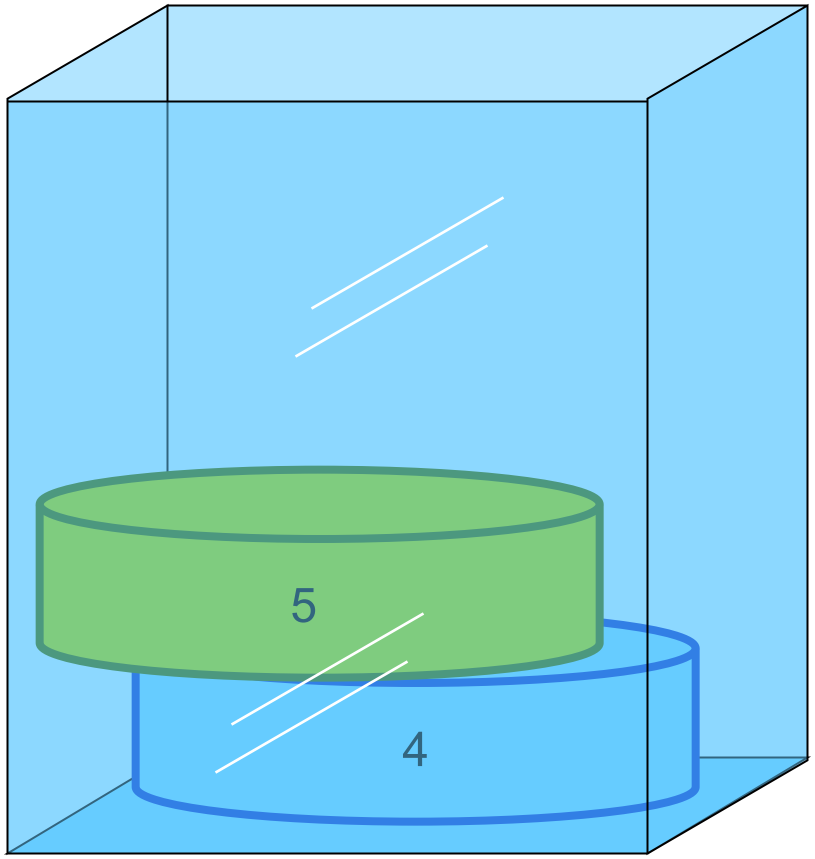
Example

4 5 + 7 2 - *



Step 2: push(5)

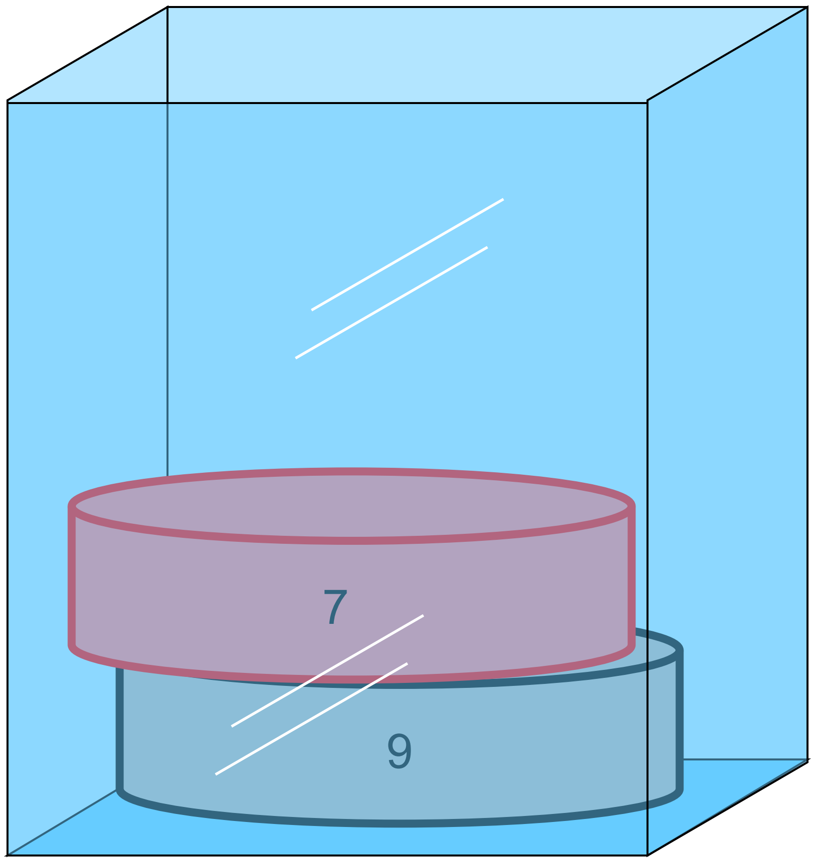
4 5 + 7 2 - *



Step 3: pop() twice and then push the result

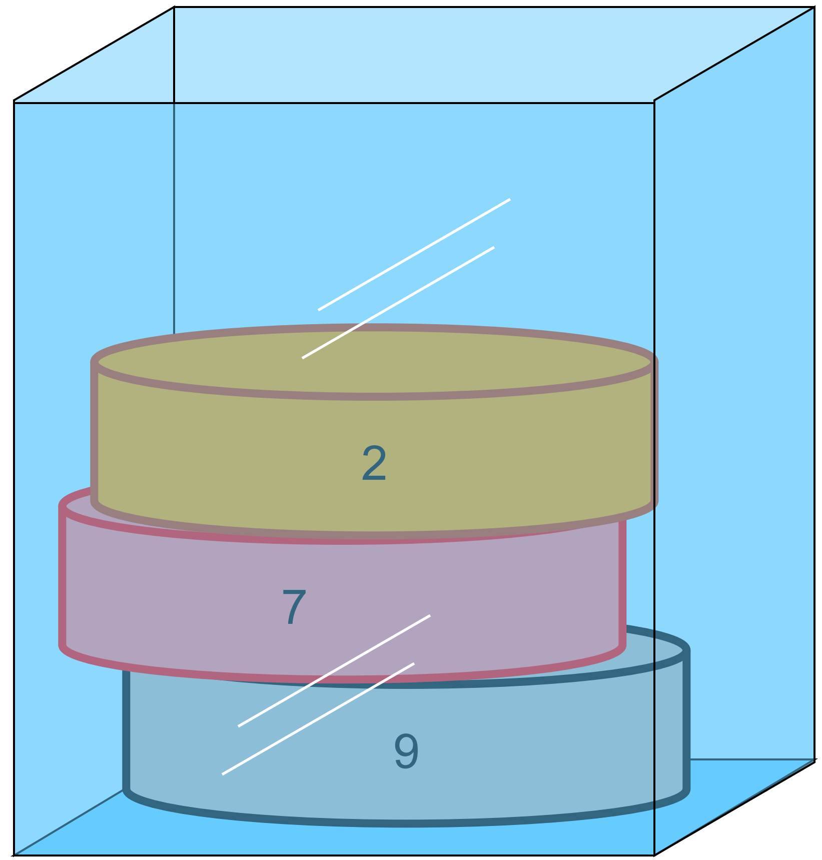
Example

4 5 + 7 2 - *



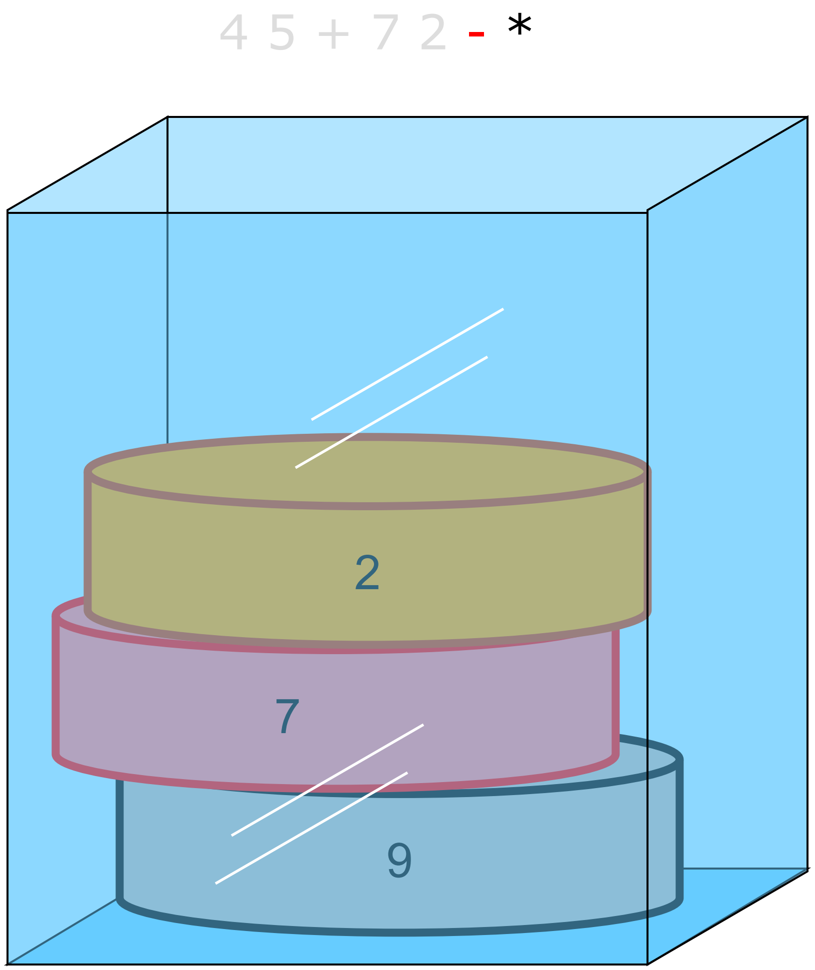
Step 4: push(7)

4 5 + 7 2 - *

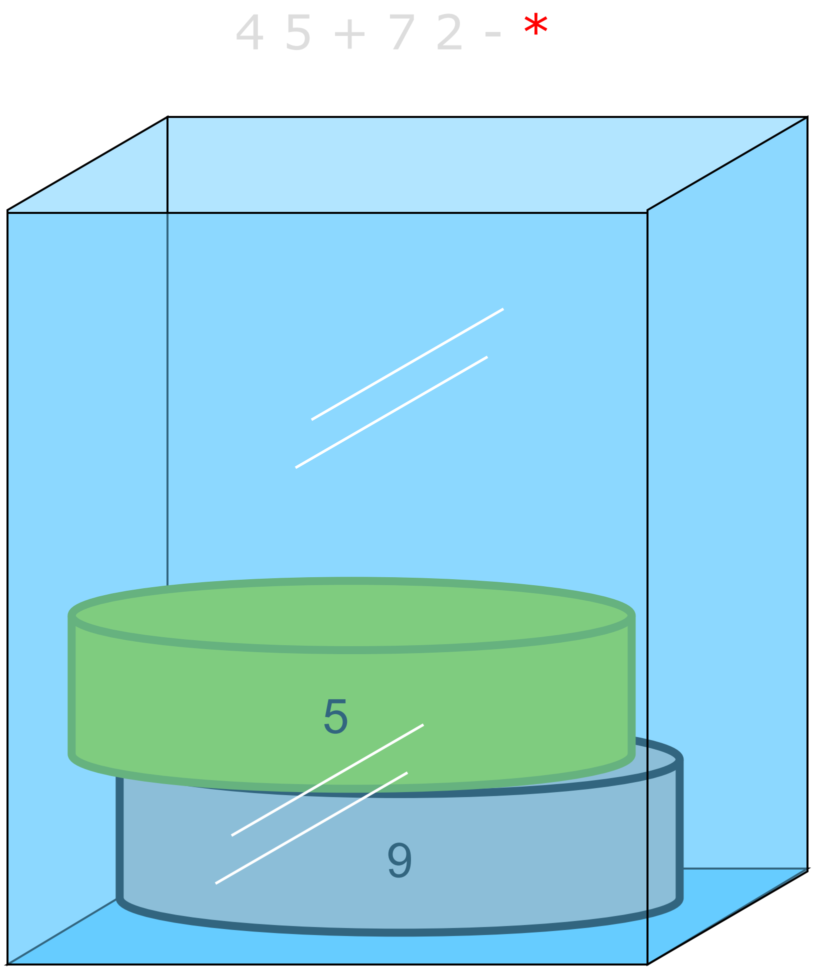


Step 5: push(2)

Example



Step 6: pop() twice and then push the result

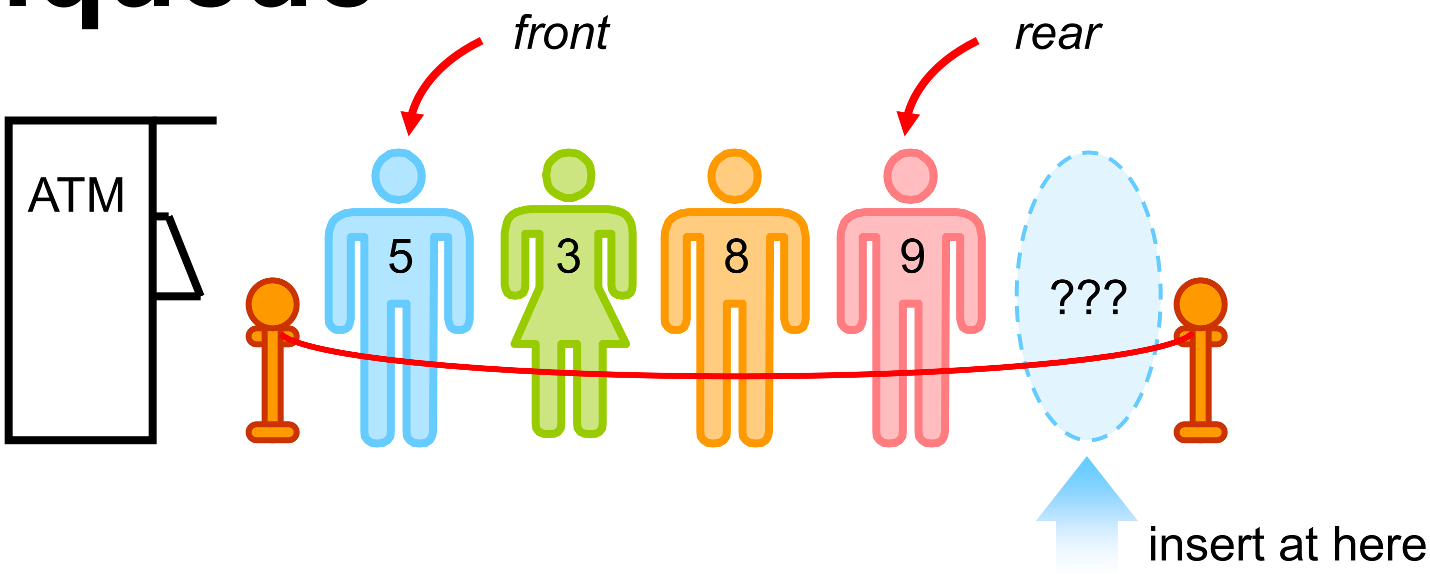


Step 7: pop() twice and compute the result

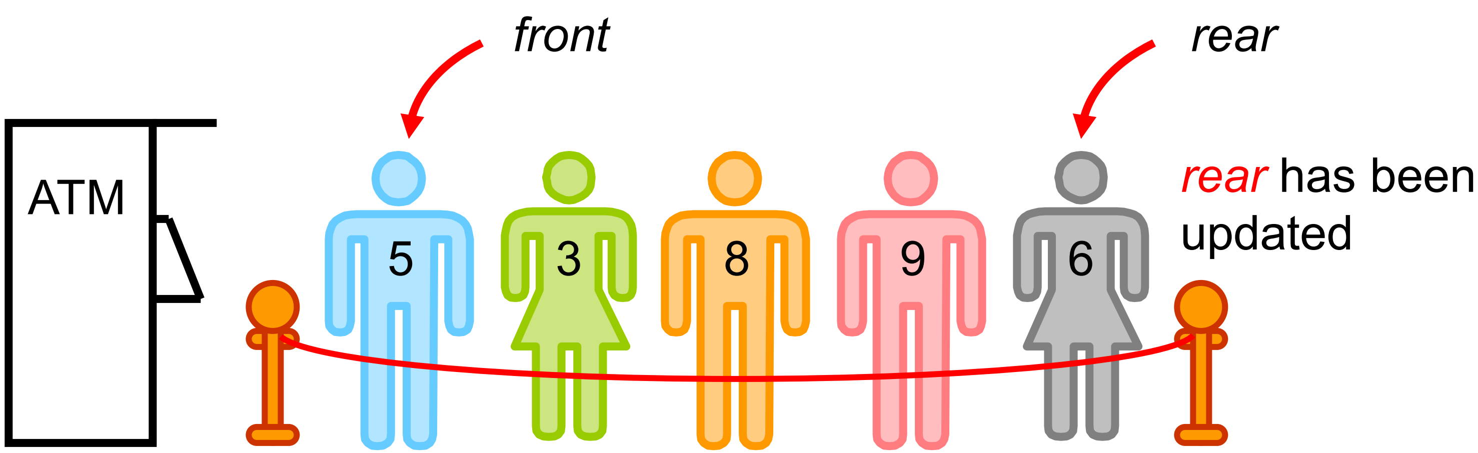
Queue

- A first-in-first-out (FIFO) queue is an ordered collection of items from which items may be deleted at one end (called the front) and into which items may be inserted at the other end (called the rear).
- Operations on a queue :
  - **initialize**: initialize the queue to an empty state
  - **size**: determine the number of elements in the queue
  - **empty**: determine if the queue is empty
  - **front**: retrieve the value of the front element
  - **back**: retrieve the value of the last element (this is not common in the applications of queue)
  - **push**: insert element at the rear of queue (in most textbooks, this operation is called enqueue)
  - **pop**: remove front element (in most textbooks, this operation is called dequeue)

Enqueue

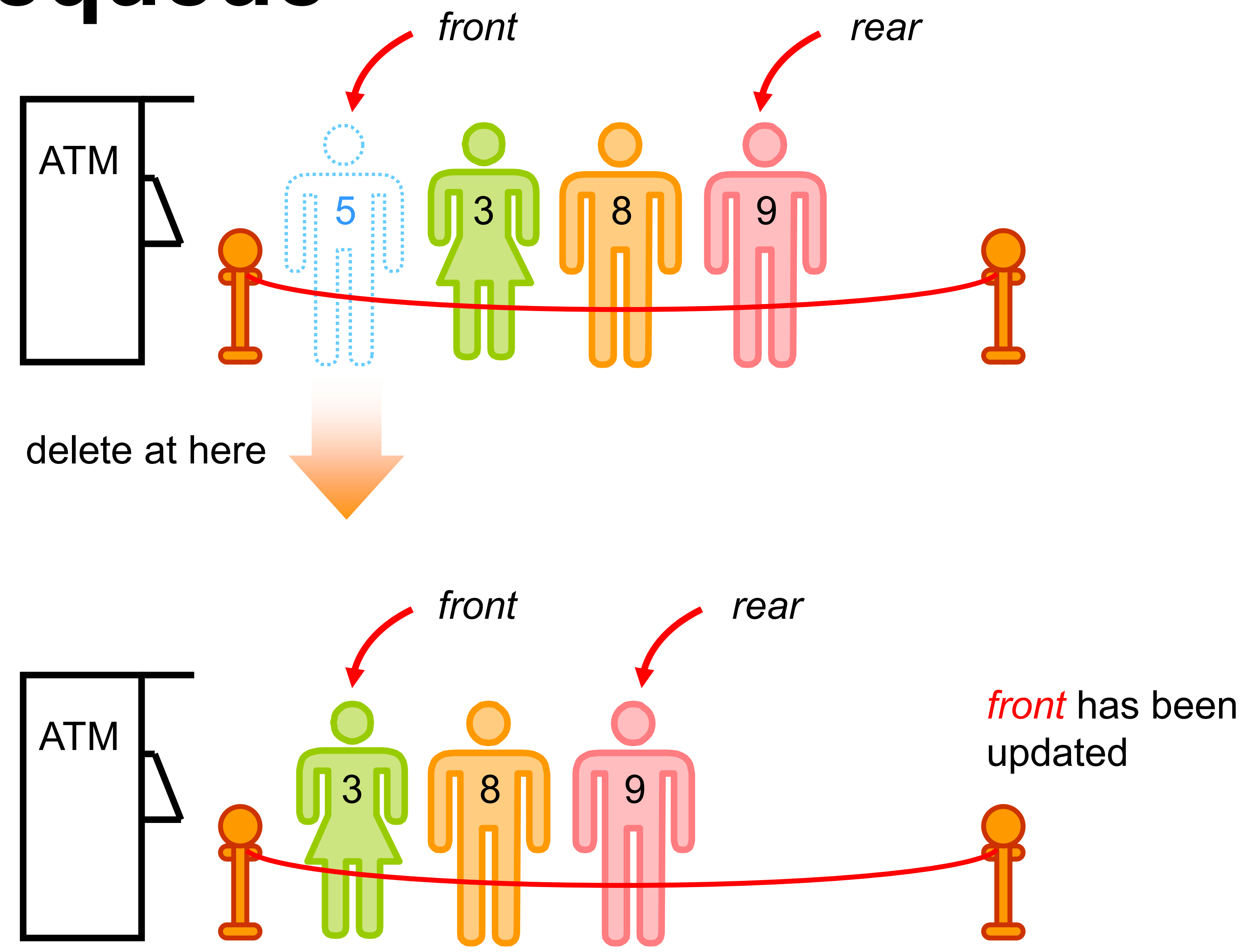


front
rear
ATM
5
3
8
9
???
insert at here



front
rear
ATM
5
3
8
9
6
rear has been updated

Dequeue


front
rear
ATM
5
3
8
9
delete at here
front
rear
ATM
3
8
9
front has been updated

Example

```
// queue::push/pop
#include <iostream>       // std::cin, std::cout
#include <queue>          // std::queue

int main ()
{
  std::queue<int> myqueue;
  int myint;

  std::cout << "Please enter some integers (enter 0 to end):\n";

  do {
    std::cin >> myint;
    myqueue.push (myint);
  } while (myint);

  std::cout << "myqueue contains: ";
  while (!myqueue.empty())
  {
    std::cout << ' ' << myqueue.front();
    myqueue.pop();
  }
  std::cout << '\n';

  return 0;
}
```

https://cplusplus.com/reference/queue/queue/push/