Lecture No.05
Stacks

CC-213 Data Structures
Department of Computer Science
University of the Punjab

Slides modified very slightly from the late Dr. Sohail Aslam’s lectures at VU

Josephus Problem

```
#include "CList.cpp"
void main(int argc, char *argv[])
{
   CList list;
   int i, N=10, M=3;
   for(i=1; i <= N; i++ ) list.add(i);

   list.start();
   while( list.length() > 1 ) {
       for(i=1; i <= M; i++ ) list.next();
       cout << "remove: " << list.get() << endl;
       list.remove();
   }
   cout << "leader is: " << list.get() << endl;
}
```

Josephus Problem

- Using a circularly-linked list made the solution trivial.

Josephus Problem

- Using a circularly-linked list made the solution trivial.
- The solution would have been more difficult if an array had been used.

Josephus Problem

- Using a circularly-linked list made the solution trivial.
- The solution would have been more difficult if an array had been used.
- This illustrates the fact that the choice of the appropriate data structures can significantly simplify an algorithm. It can make the algorithm much faster and efficient.

Josephus Problem

- Using a circularly-linked list made the solution trivial.
- The solution would have been more difficult if an array had been used.
- This illustrates the fact that the choice of the appropriate data structures can significantly simplify an algorithm. It can make the algorithm much faster and efficient.
- Later we will see how some elegant data structures lie at the heart of major algorithms.

Josephus Problem

- Using a circularly-linked list made the solution trivial.
- The solution would have been more difficult if an array had been used.
- This illustrates the fact that the choice of the appropriate data structures can significantly simplify an algorithm. It can make the algorithm much faster and efficient.
- Later we will see how some elegant data structures lie at the heart of major algorithms.
- An entire CS course “Design and Analysis of Algorithms” is devoted to this topic.

Abstract Data Type

- We have looked at four different implementations of the List data structures:
  - Using arrays
  - Singly linked list
  - Doubly linked list
  - Circularly linked list.

Abstract Data Type

- We have looked at four different implementations of the List data structures:
  - Using arrays
  - Singly linked list
  - Doubly linked list
  - Circularly linked list.
- The interface to the List stayed the same, i.e., add(), get(), next(), start(), remove() etc.

Abstract Data Type

- We have looked at four different implementations of the List data structures:
  - Using arrays
  - Singly linked list
  - Doubly linked list
  - Circularly linked list.
- The interface to the List stayed the same, i.e., add(), get(), next(), start(), remove() etc.
- The list is thus an abstract data type; we use it without being concerned with how it is implemented.

Abstract Data Type

- What we care about is the methods that are available for use with the List ADT.

Abstract Data Type

- What we care about is the methods that are available for use with the List ADT.
- We will follow this theme when we develop other ADT.

Abstract Data Type

- What we care about is the methods that are available for use with the List ADT.
- We will follow this theme when we develop other ADT.
- We will publish the interface and keep the freedom to change the implementation of ADT without effecting users of the ADT.

Abstract Data Type

- What we care about is the methods that are available for use with the List ADT.
- We will follow this theme when we develop other ADT.
- We will publish the interface and keep the freedom to change the implementation of ADT without effecting users of the ADT.
- The C++ classes provide us the ability to create such ADTs.

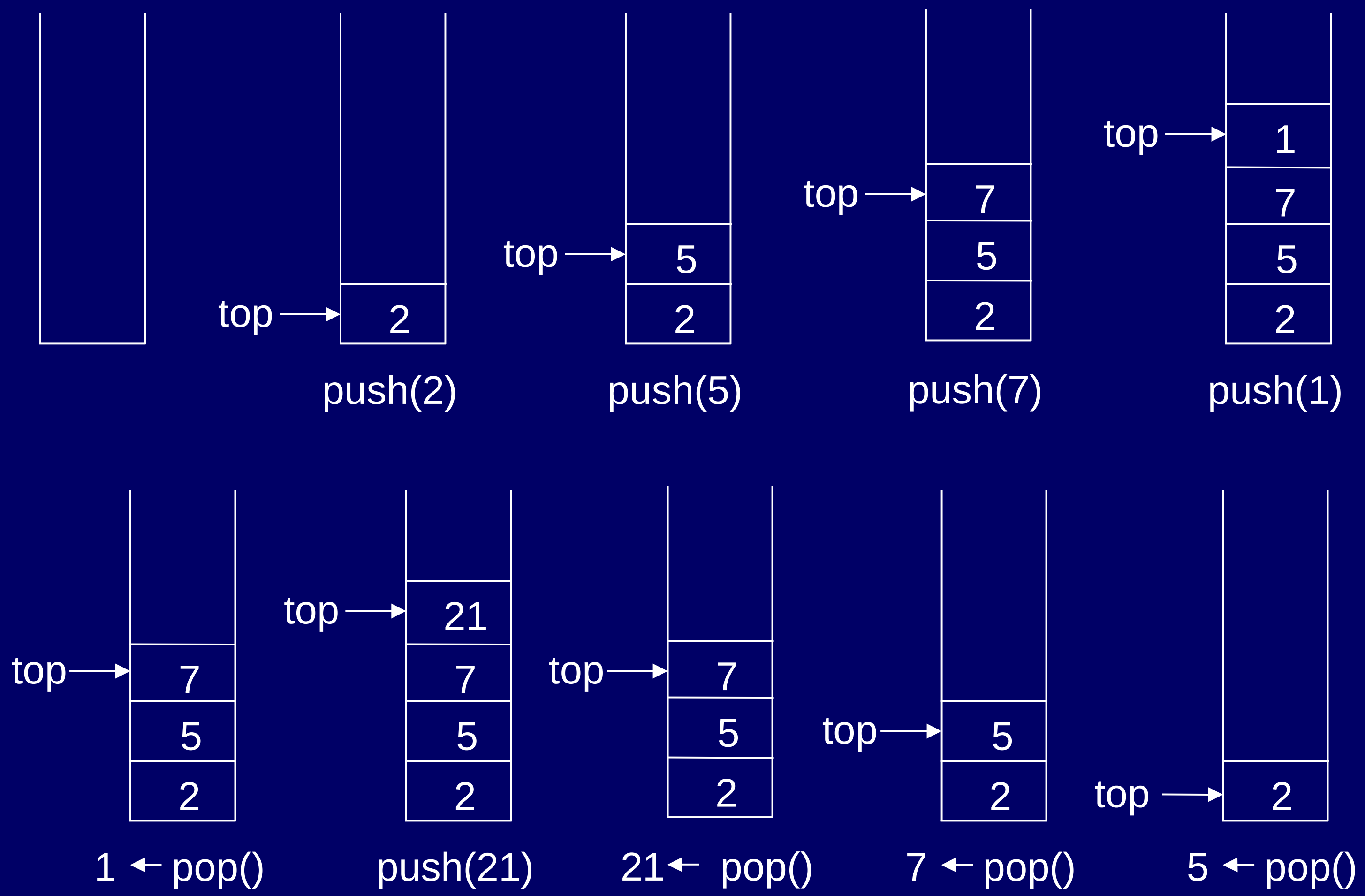
Stacks

- Stacks in real life: stack of books, stack of plates
- Add new items at the top
- Remove an item at the top
- Stack data structure similar to real life: collection of elements arranged in a linear order.
- Can only access element at the top

Stack Operations

- Push(X) – insert X as the top element of the stack
- Pop() – remove the top element of the stack and return it.
- Top() – return the top element without removing it from the stack.

Stack Operations



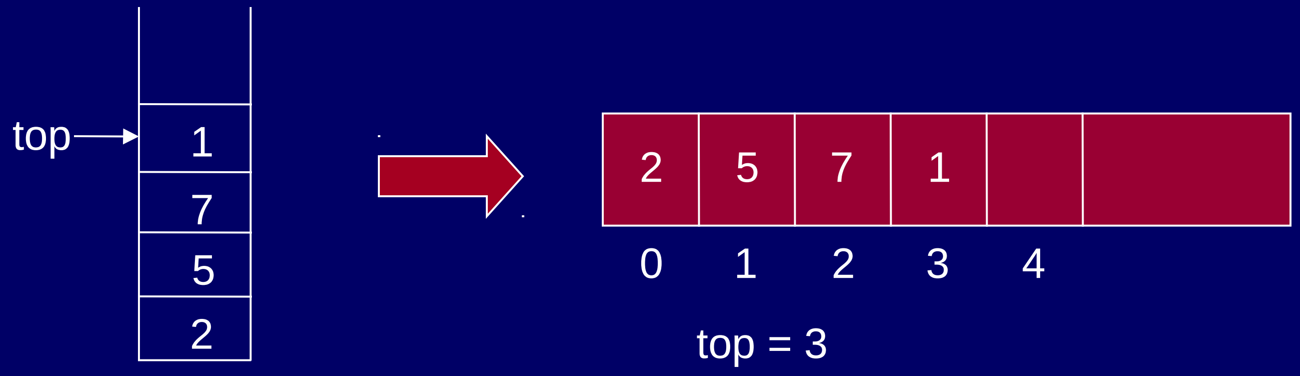
Stack Operation

- The last element to go into the stack is the first to come out: *LIFO* – Last In First Out.
- What happens if we call pop() and there is no element?
- Have IsEmpty() boolean function that returns true if stack is empty, false otherwise.
- Throw StackEmpty exception: advanced C++ concept.

Stack Implementation: Array

- Worst case for insertion and deletion from an array when insert and delete from the beginning: shift elements to the left.
- Best case for insert and delete is at the end of the array – no need to shift any elements.
- Implement push() and pop() by inserting and deleting at the end of an array.

Stack using an Array



Stack using an Array

- In case of an array, it is possible that the array may “fill-up” if we push enough elements.
- Have a boolean function **`IsFull()`** which returns true is stack (array) is full, false otherwise.
- We would call this function before calling push(x).

Stack Operations with Array

```
int pop()
{
    return A[current--];
}

void push(int x)
{
    A[++current] = x;
}
```

Stack Operations with Array

```
int top()
{
    return A[current];
}
int IsEmpty()
{
    return ( current == -1 );
}
int IsFull()
{
    return ( current == size-1);
}
```

- A quick examination shows that all five operations take constant time.

Stack Using Linked List

- We can avoid the size limitation of a stack implemented with an array by using a linked list to hold the stack elements.
- As with array, however, we need to decide where to insert elements in the list and where to delete them so that push and pop will run the fastest.

Stack Using Linked List

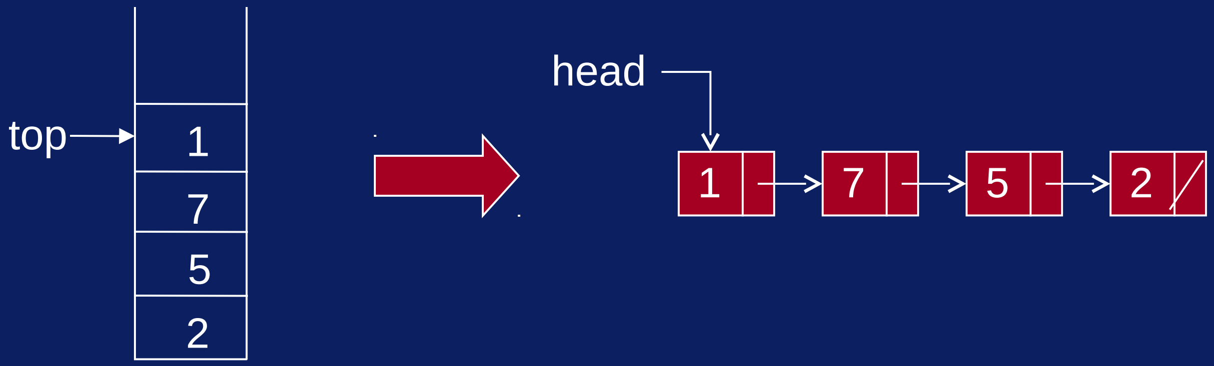
- We can avoid the size limitation of a stack implemented with an array by using a linked list to hold the stack elements.
- As with array, however, we need to decide where to insert elements in the list and where to delete them so that push and pop will run the fastest.

Stack Using Linked List

- For a singly-linked list, insert at start or end takes constant time using the head and current pointers respectively.
- Removing an element at the start is constant time but removal at the end required traversing the list to the node one before the last.
- Make sense to place stack elements at the start of the list because insert and removal are constant time.

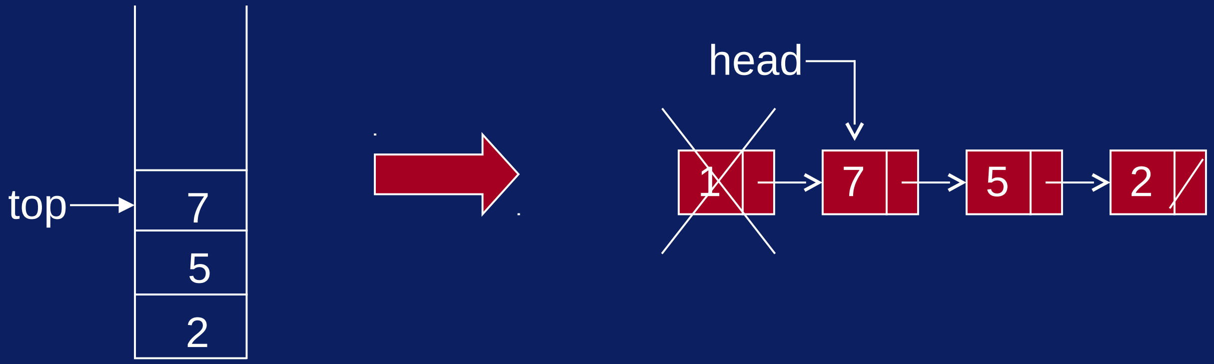
Stack Using Linked List

- No need for the current pointer; head is enough.



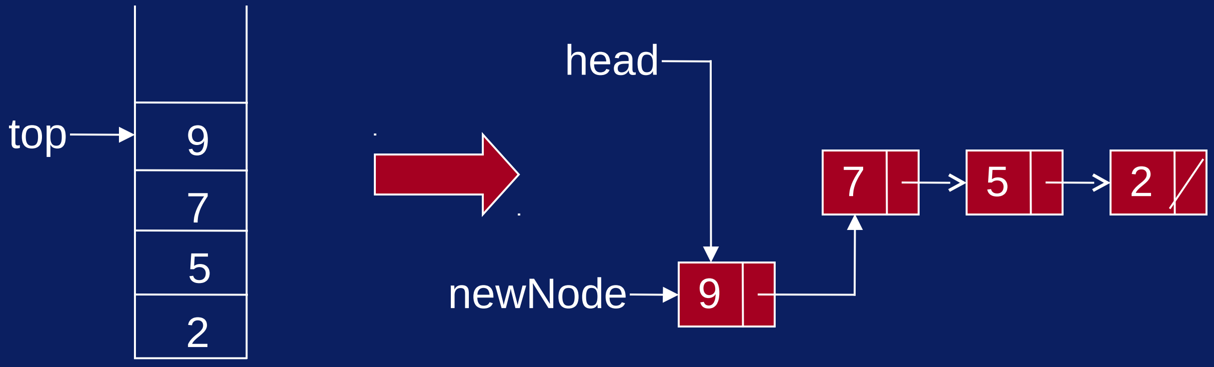
Stack Operation: List

```
int pop()
{
    int x = head->get();
    Node* p = head;
    head = head->getNext();
    delete p;
    return x;
}
```



Stack Operation: List

```
void push(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(head);
    head = newNode;
}
```



push(9)

Stack Operation: List

```
int top()
{
    return head->get();
}
int IsEmpty()
{
    return ( head == NULL );
}
```

- All four operations take constant time.

Stack: Array or List

- Since both implementations support stack operations in constant time, any reason to choose one over the other?
- Allocating and deallocating memory for list nodes does take more time than preallocated array.
- List uses only as much memory as required by the nodes; array requires allocation ahead of time.
- List pointers (head, next) require extra memory.
- Array has an upper limit; List is limited by dynamic memory allocation.