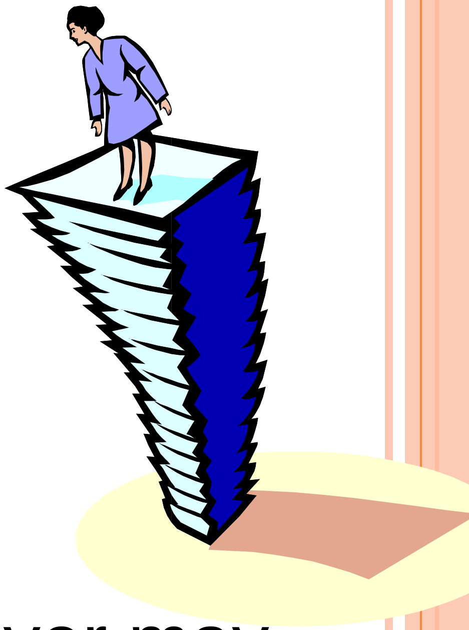
Stack

INTRODUCTION TO STACKS



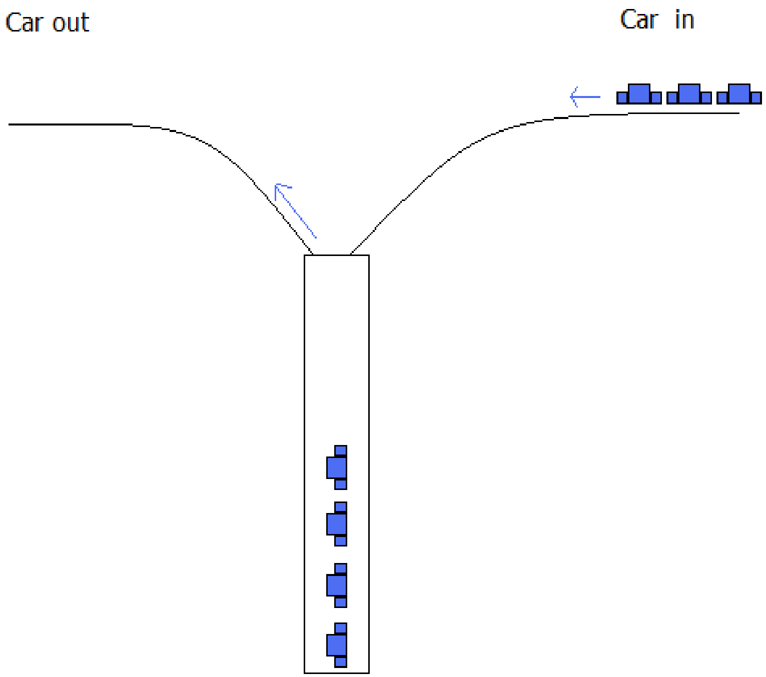
- Consider the following problems:

Problem 1:

For a poker game; on any turn, a player may *discard a single card* from his hand to the top of the pile, or he may *retrieve the top card* from the discard pile

Is there an appropriate data type to model this discard pile???

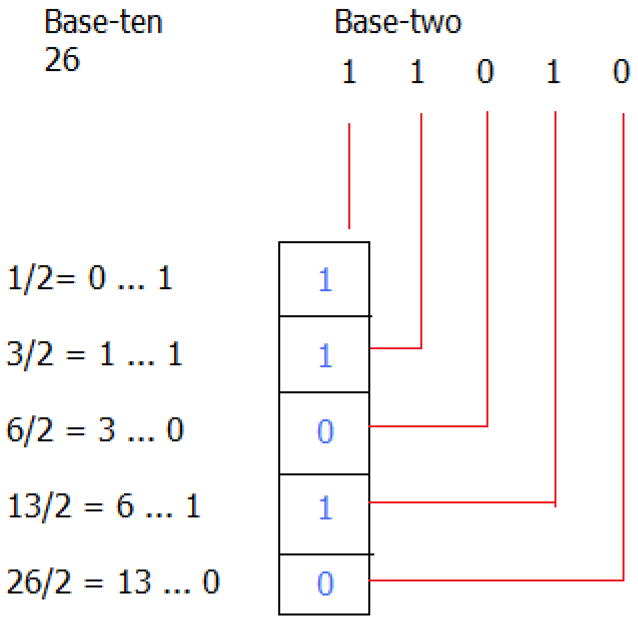
Introduction to Stacks



Is there an appr el this parking lot???

INTRODUCTION TO STACKS

- An algorithm converting 26 (11010) into base-two representation



Introduction to Stacks

- Each problem involves a collection of related data items:

1. The basic operations are adding a card to and removing a card from the top of discard pile
2. The basic operation are pushing a car onto the parking lot and removing the last car previously placed on the parking lot
3. We notice that the remainders are generated in reverse order (right to left), therefore, they must be stored in some structure so they can later be displayed in the usual left-to-right order

Introduction to Stacks

- This type of last-in-first-out processing occurs in a wide variety of applications
- This last-in-first-out (LIFO) data structure is called a **Stack**
- Adding an item to a stack is referred to as **pushing** that item onto the stack
- Removing an item from the stack is referred to as **popping** the stack

Designing and Building a Stack class

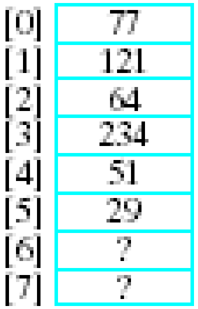
- The basic functions are:
  - Constructor: construct an empty stack
  - Empty(): Examines whether the stack is empty or not
  - Push(): Add a value at the top of the stack
  - Top(): Read the value at the top of the stack
  - Pop(): Remove the value at the top of the stack
  - Display(): Displays all the elements in the stack

Selecting storage structures

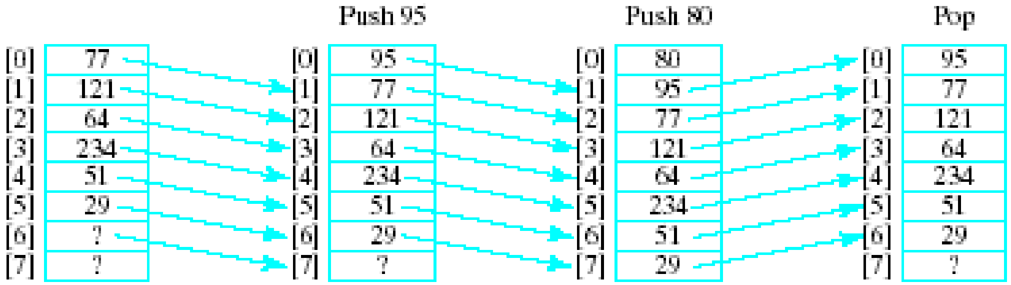
- Two choices
  - Select position 0 as top of the stack
  - Select position 0 as bottom of the stack

SELECT POSITION 0 AS TOP OF THE STACK

- Model with an array
  - Let position 0 be top of stack

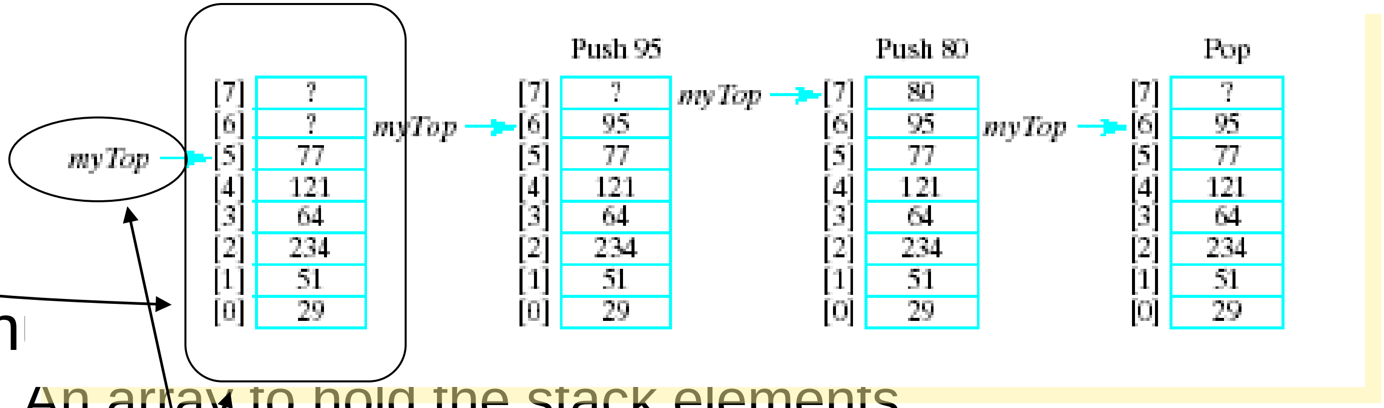


- Problem … consider pushing and popping
  - Requires much shifting



SELECT POSITION 0 AS BOTTOM OF THE STACK

- A better approach is to let position 0 be the bottom of the stack



		Push 95	Push 80	Pop
[7]	?	?	80	?
[6]	?	95	95	95
[5]	77	77	77	77
[4]	121	121	121	121
[3]	64	64	64	64
[2]	234	234	234	234
[1]	51	51	51	51
[0]	29	29	29	29

- Th
  - An array to hold the stack elements
  - An integer to indicate the top of the stack

Implementation of the Operations

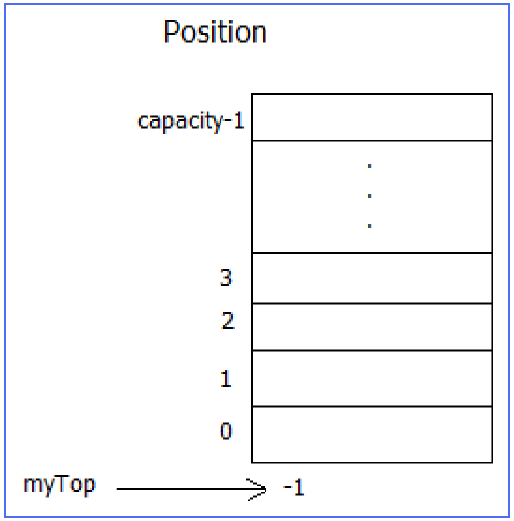
- Constructor:

  Create an array: (int) array[capacity]

  Set myTop = -1

- Empty():

  check if myTop == -1



Implementation of the Operations

- Push(int x):

```
if array is not FULL (myTop < capacity-1)
    myTop++
    store the value x in array[myTop]
else
    output “out of space”
```

Implementation of the Operations

- Top():

```
If the stack is not empty
        return the value in array[myTop]
else:
        output “no elements in the stack”
```

Implementation of the Operations

- Pop():

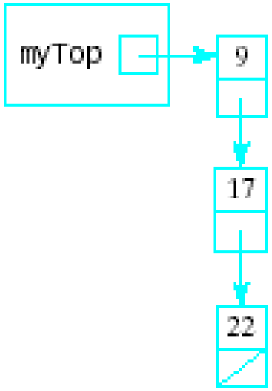
```
If the stack is not empty
    myTop -= 1
else:
    output “no elements in the stack”
```

Further Considerations

- What if static array initially allocated for stack is too small?
  - Terminate execution?
  - Replace with larger array!
- Creating a larger array
  - Allocate larger array
  - Use loop to copy elements into new array
  - Delete old array

Linked Stacks

- Another alternative to allowing stacks to grow as needed
- Linked list stack needs only one data member
  - Pointer `myTop`
  - Nodes allocated (but not part of stack class)

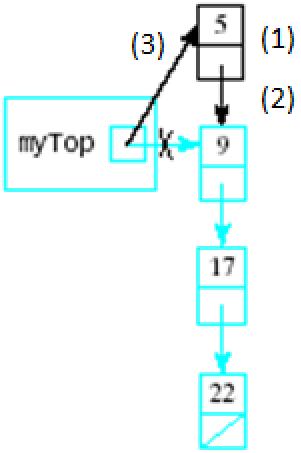


Implementing Linked Stack Operations

- Constructor
  - Simply assign null pointer to `myTop`
- Empty
  - Check for `myTop == null`
- Push
  - Insertion at beginning of list

```
myTop == new stack::Node(value,
  mytop)
```

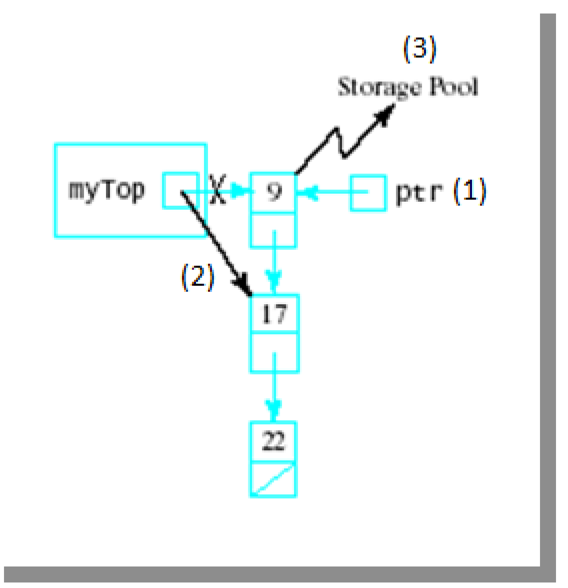
- Top
  - Return data to which `myTop` points



Implementing Linked Stack Operations

- Pop
  - Delete first node in the linked list

```
ptr = myTop;
myTop = myTop->next;
delete ptr;
```



- Output
  - Traverse the list

```
for (ptr = myTop;
      ptr != 0; ptr = ptr->next)
   out << ptr->data << endl;
```

C/C++ Standard library

- The **C standard library** (also known as **libc**) is a now-standardized collection of header files and library routines used to implement common operations, such as input/output and string handling

- For example:

  #include <iostream>

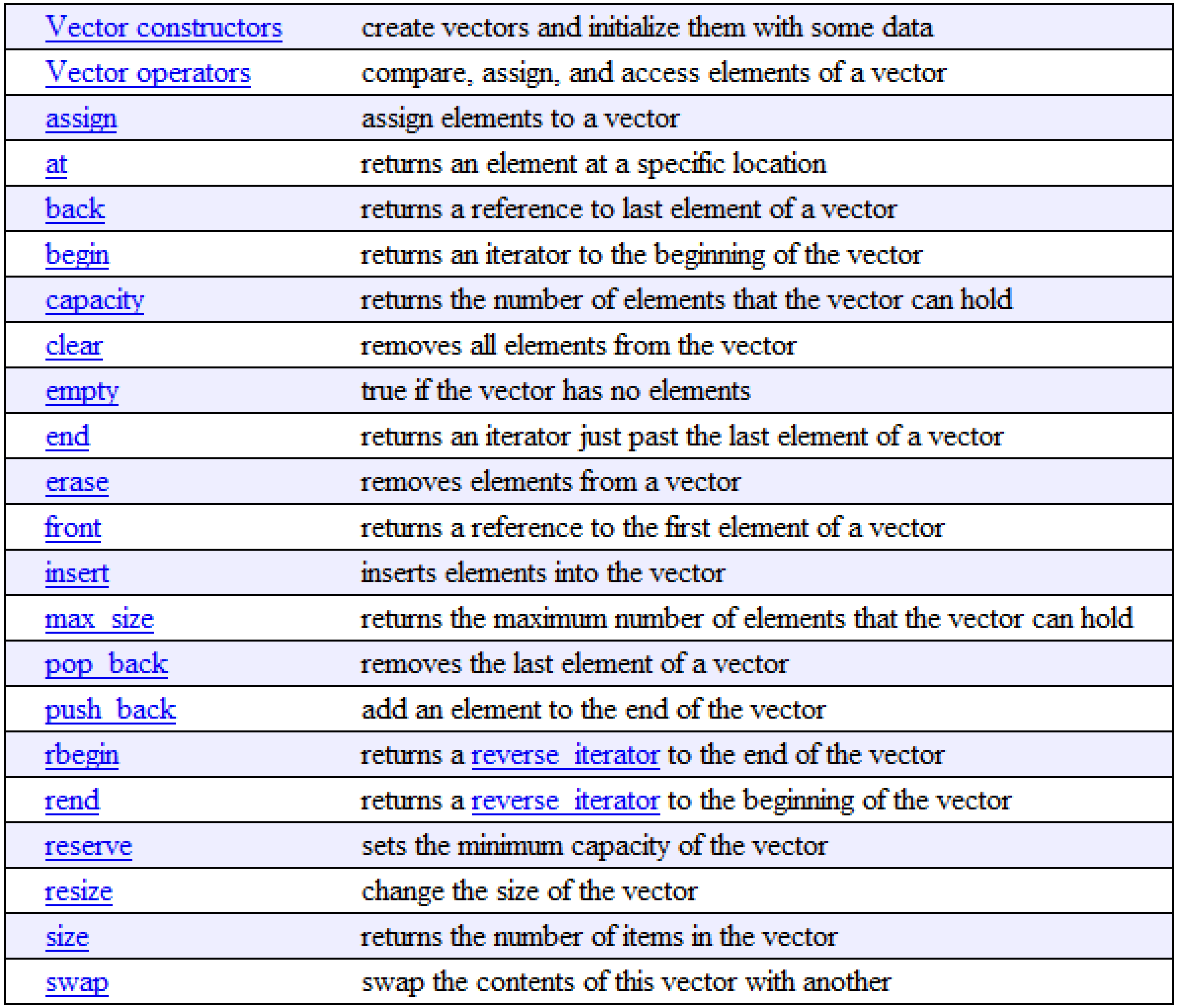
VECTOR

- Vectors contain contiguous elements stored as an **Dynamic** array.

- All you have to do is include vector from library

  #include <vector>

Vector Functions



Vector constructors	create vectors and initialize them with some data
Vector operators	compare, assign, and access elements of a vector
assign	assign elements to a vector
at	returns an element at a specific location
back	returns a reference to last element of a vector
begin	returns an iterator to the beginning of the vector
capacity	returns the number of elements that the vector can hold
clear	removes all elements from the vector
empty	true if the vector has no elements
end	returns an iterator just past the last element of a vector
erase	removes elements from a vector
front	returns a reference to the first element of a vector
insert	inserts elements into the vector
max_size	returns the maximum number of elements that the vector can hold
pop_back	removes the last element of a vector
push_back	add an element to the end of the vector
rbegin	returns a reverse_iterator to the end of the vector
rend	returns a reverse_iterator to the beginning of the vector
reserve	sets the minimum capacity of the vector
resize	change the size of the vector
size	returns the number of items in the vector
swap	swap the contents of this vector with another

Designing and Building a Stack class

- The basic functions are:
  - Constructor: construct an empty stack
  - Empty(): Examines whether the stack is empty or not
  - Push(): Add a value at the top of the stack
  - Top(): Read the value at the top of the stack
  - Pop(): Remove the value at the top of the stack
  - Display(): Displays all the elements in the stack

Functions related to Stack

- Constructor: vector<int> L;
- Empty(): L.size() == 0?
- Push(): L.push_back(value);
- Top(): L.back();
- Pop(): L.pop_back();
- Display(): Write your own

A Small Example

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


char name[20];
int i, j, k;

int main()
{

    vector<int> L;
    L.push_back(1);
    L.push_back(2);
    L.push_back(3);
    L.pop_back();

    for(i=0;i<L.size();i++)
            cout << L[i] << " ";

    cout << L.back();

    cin >> name;
}
```

Use of Stack in Function calls

- Whenever a function begins execution, an **activation record** is created to store the **current environment** for that function

- Current environment includes the
  - values of its parameters,
  - contents of registers,
  - the function’s return value,
  - local variables
  - address of the instruction to which execution is to **return** when the function finishes execution (If execution is interrupted by a call to another function)

Use of Stack in Function calls

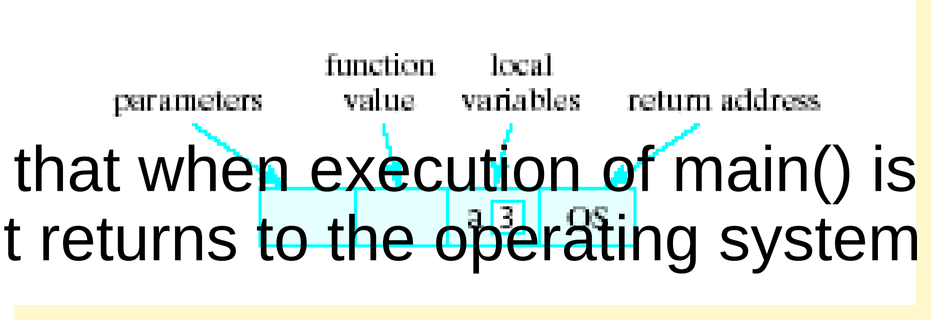
- Functions may call other functions and thus interrupt their own execution, some data structure must be used to store these activation records so they can be recovered and the system can be reset when a function resumes execution
- It is the fact that the last function interrupted is the first one reactivated
- It suggests that a stack can be used to store these activation records
- A stack is the appropriate structure, and since it is manipulated during execution, it is called the **run-time stack**

Consider the following program segment

```
int main(){
     int a=3;
     f1(a);
     cout << endl;
}

Void f1(int x){
     cout << f2(x+1);
}

Int f2(int p){
     int q=f3(p/2);
     return 2*q;
}

Int f3(int n){
     return n*n+1;
}
```

Run-time Stack

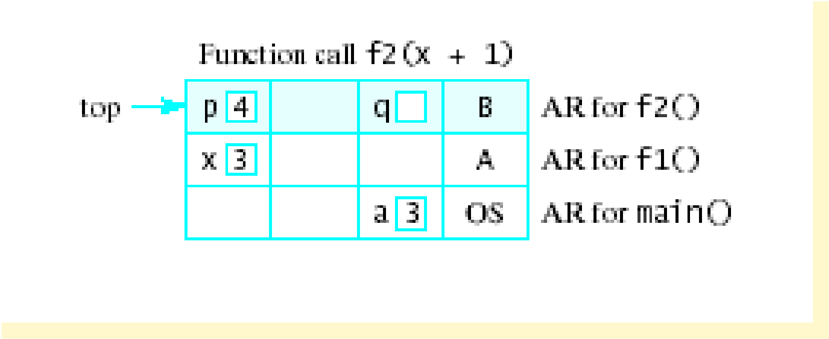
- OS denotes that when execution of main() is completed, it returns to the operating system



Use of Run-time Stack

When a function is called …

- Copy of activation record pushed onto run-time stack
- Arguments copied into parameter spaces
- Control transferred to starting address of body of function



Function call f2(x + 1)

top →	p 4		q □	B	AR for f2()
	x 3			A	AR for f1()
			a 3	OS	AR for main()