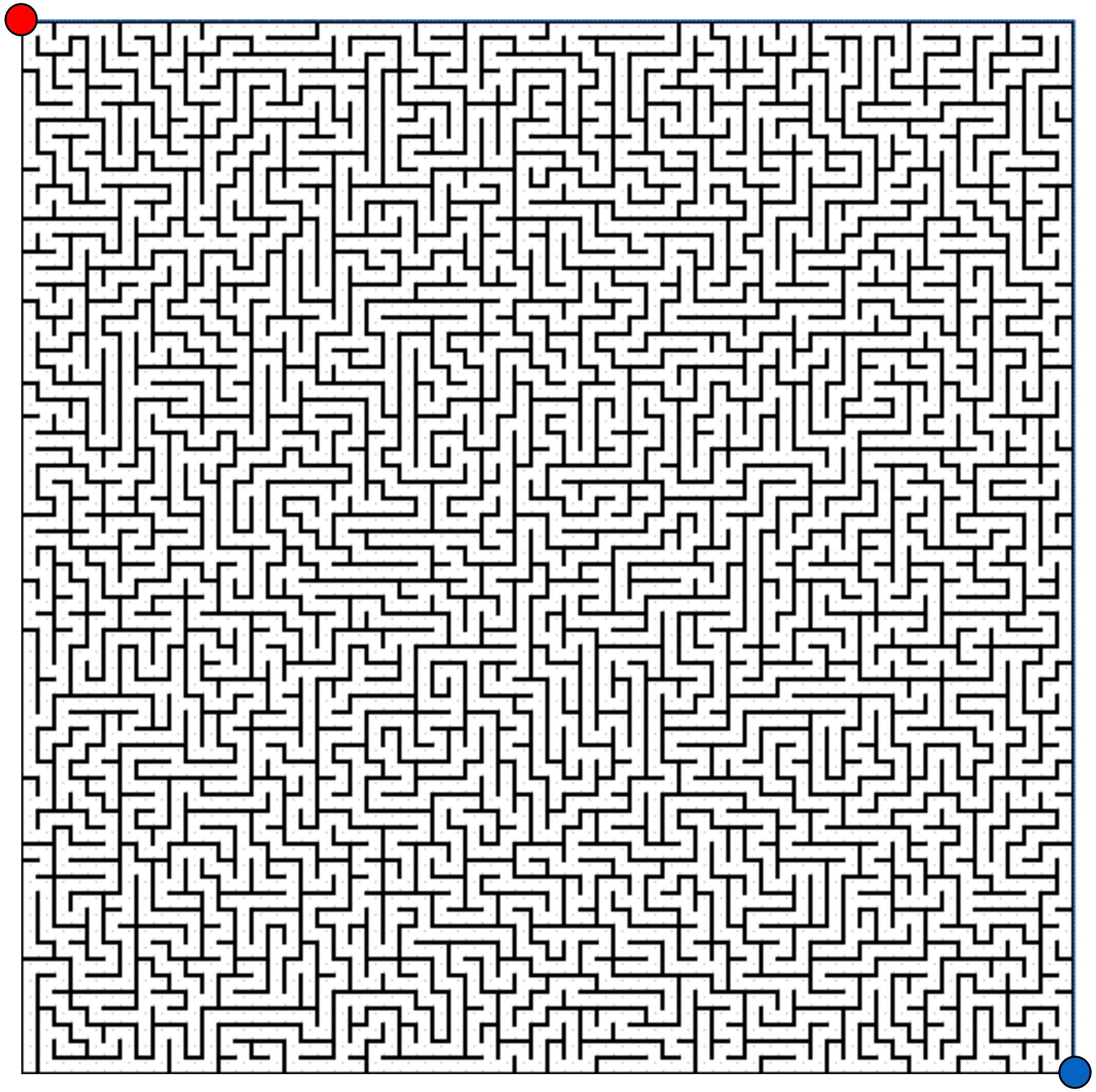
CS3334 Data Structures
Lec-2 Stacks

Generating and Solving Maze



Review about Linked Lists

- Abstract Data Types
- Pointers and References
- Singly Linked List
- Circular Lists
- Doubly Linked Lists
- Applications

Review: Abstract Data Type

```
/*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();
…
};
```

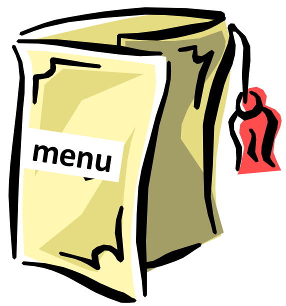
Some advanced data types are very useful.

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/she looks at what the tools can do. i.e. He/she looks at the **abstract data types (ADT).**

He/She needs not know how the tools have been implemented.



Review: Abstract Data Type

e. g. The ***set*** ADT:

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.

Review: 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.

…

Review: Passing Arguments

Pass by Value

```
int SquareByV(int a)
{
    Return a * a;
}
int main()
{
    int x =2;
    SquareByV(x);
}
```

Pass by Pointer

```
void SquareByP(int *cptr)
{
    *cptr *= *cptr;
}
int main()
{
    int y =2;
    SquareByP(&y);
}
```

Pass by Reference

```
void SquareByR(int &cRef)
{
    cRef *=cRef;
}
int main()
{
    int z =2;
    SquareByR(z);
}
```

int * p : (int *) p

*p: dereference (the data pointed by p)

*: multiplication

&a: the address of a

A[]: array variable is a pointer

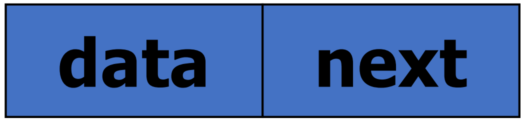
Review: List

A Linear List (or **a list**, for short)

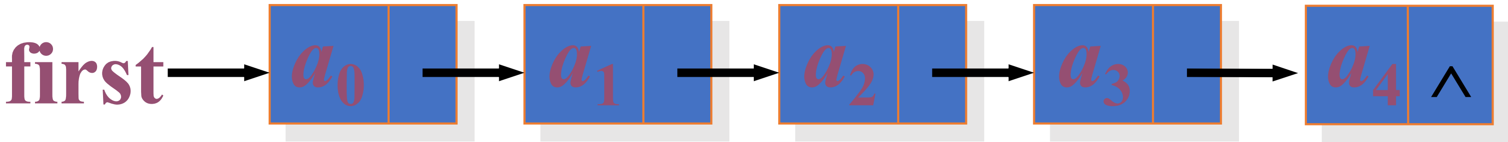
- is a sequence of n nodes $\{x_1, x_2, .., x_n\}$ whose essential structural properties involve only the relative positions between items as they appear in a line.
- can be implemented as
  - Arrays: statically allocated or dynamically allocated
  - Linked Lists: dynamically allocated
- A list can be sorted or unsorted.

Review: Singly Linked List

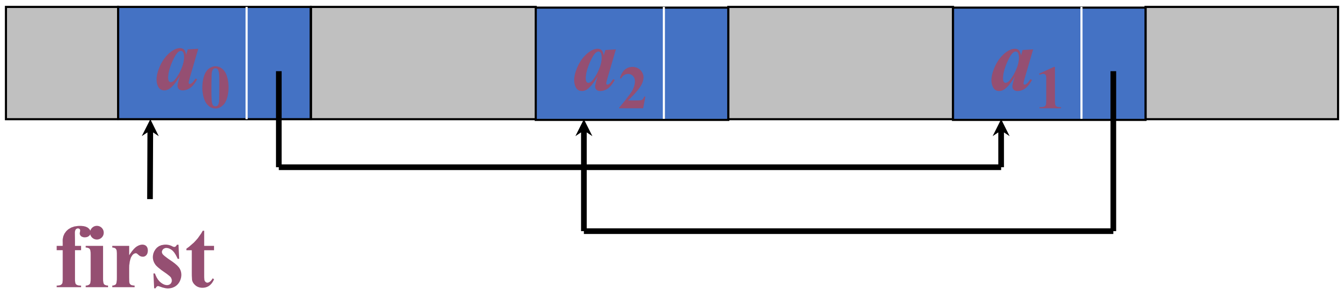
- Each item in the list is a node



- Linear Structure



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



Review: Singly Linked List

```
// 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;
}
```

Review: Singly Linked List

- Operations:
  - InsertNode: insert a new node into a list
  - RemoveNode: remove a node from a list
  - SearchNode: search a node in a list
  - CountNodes: compute the length of a list
  - PrintContent: print the content of a list
  - …
- All the variables are defined to be “int”, how about when we want to use “double”?
  - Write a different class of list for “double”? Or…

Review: Search for a node

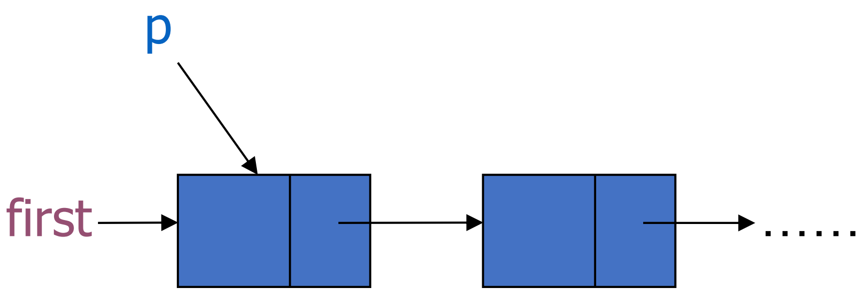
Use a pointer p to traverse the list

- If found: return the pointer to the node,
- otherwise: return NULL.

```
//List.cpp
ListNode* List::Search(int data)
{
    ListNode *p=first;
    while (p!=NULL)
    {
        if (p->getData()==data)
            return p;
        p=p->getNext();
    }
    return NULL;
}
```

```
// 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;
}
```



Review: Insert a node

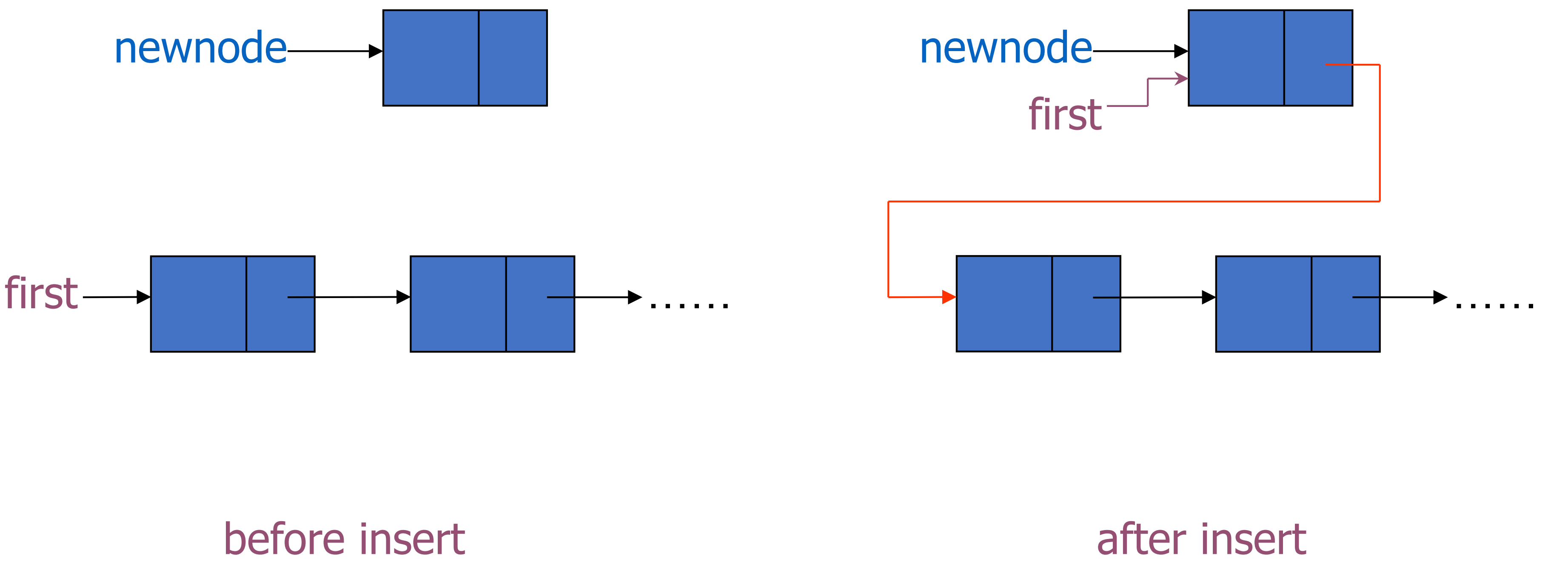
- Data comes one by one, and we want to keep them sorted, how?
  - Do search to locate the position
  - Insert the new node
- Why is this correct?

- We will encounter three cases of insert position
  - Insert before the first node
  - Insert in the middle
  - Insert at the end of the list
- One important case missing: Empty List

```
Void List::InsertNode(ListNode* newnode)
{
        if (first==NULL)
                first=newnode;
        else if (newnode->getData() < first->getData() )
                …
}
```

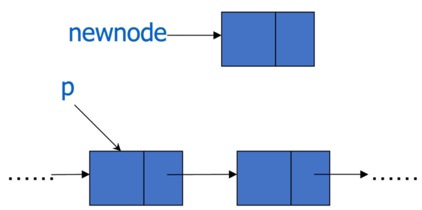
Review: Insert a node: Case 1

- Insert before the first node
  - newnode->next=first
  - first=newnode

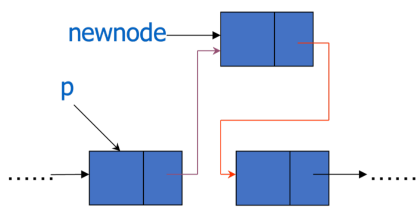


Review: Insert a node: Case 2 & 3

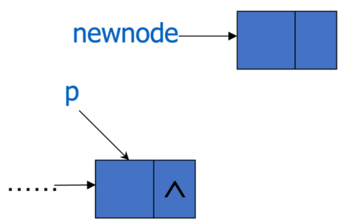
- Insert in the middle or at the end of the list
  - newnode->next = p->next
  - p->next = newnode



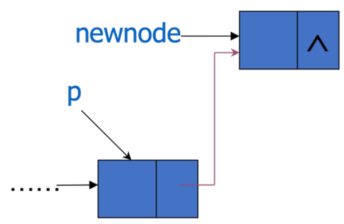
before insert



after insert



before insert



after insert

Review: Insert a node: Complete Solution

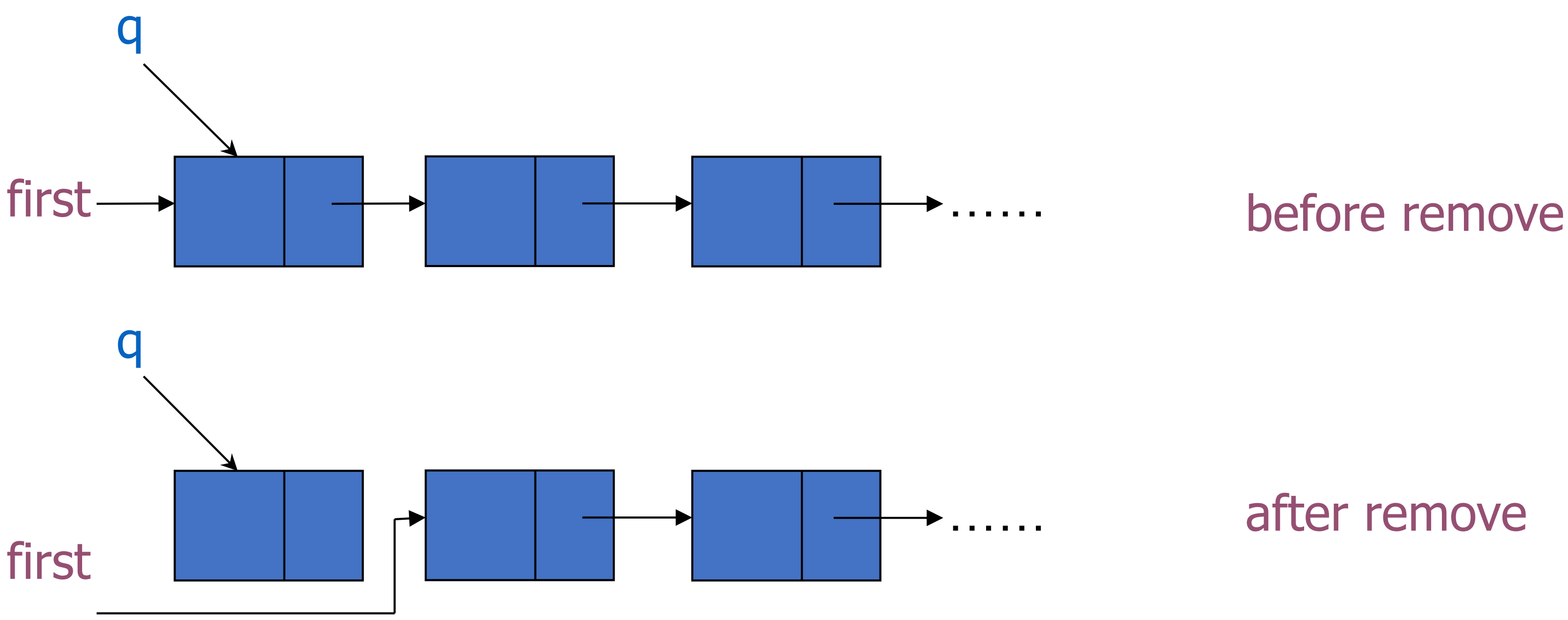
```
Void List::InsertNode(ListNode* newnode)
{
        if (first == NULL)
                first = newnode;
        else if (newnode->data < first->data) {
                newnode->next = first;
                first = newnode;
        }
        else {
                ListNode* p=first;
                while(p->next != NULL && newnode->data > p->next->data)
                        p = p->next;
                //p will stop at the last node or at the node
                //after which we should insert the new node
                //note that p->next might be NULL here
                newnode->next = p->next;
                p->next = newnode;
        }
}
```

Review: 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

Review: Remove a node: Case 1

- Remove a node at the beginning of the list
  - Current status: the node pointed by "first" is unwanted
  - The action we need: q=first; first=q->next

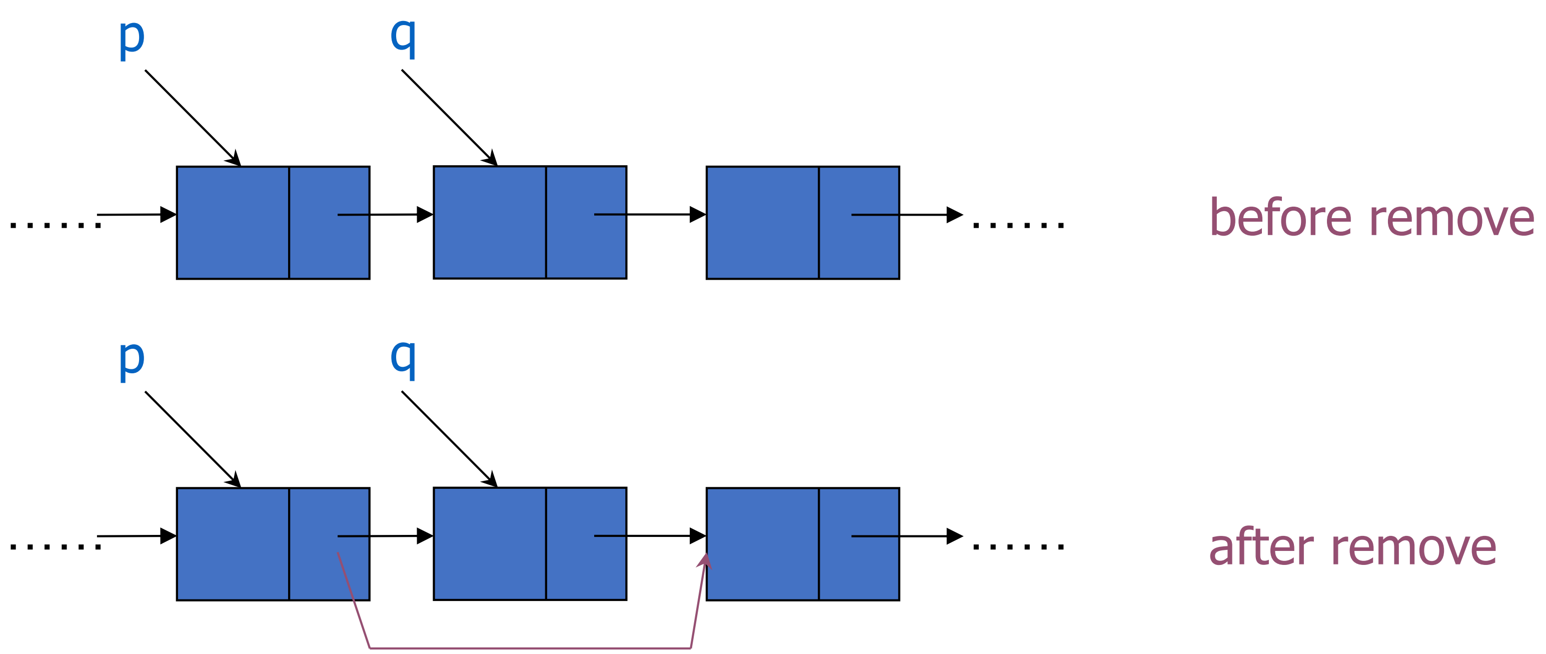


Review: Remove a node: Case 2

- Remove a node not at the beginning of the list

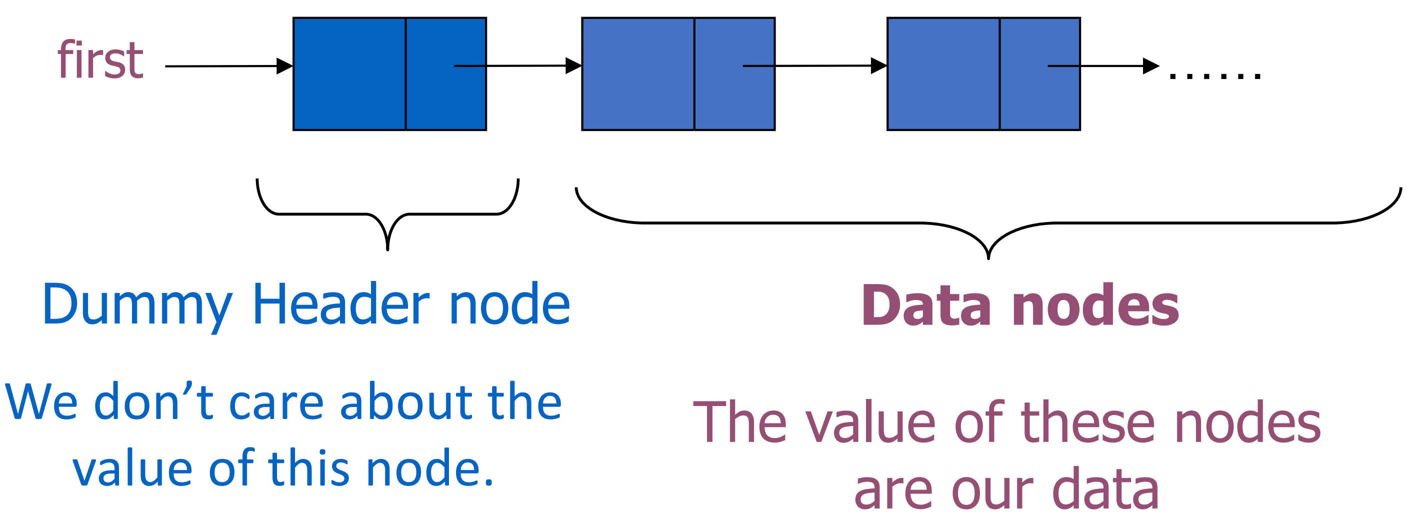
  Current status: q == p->next and the node pointed by q is unwanted

  The action we need: p->next=q->next



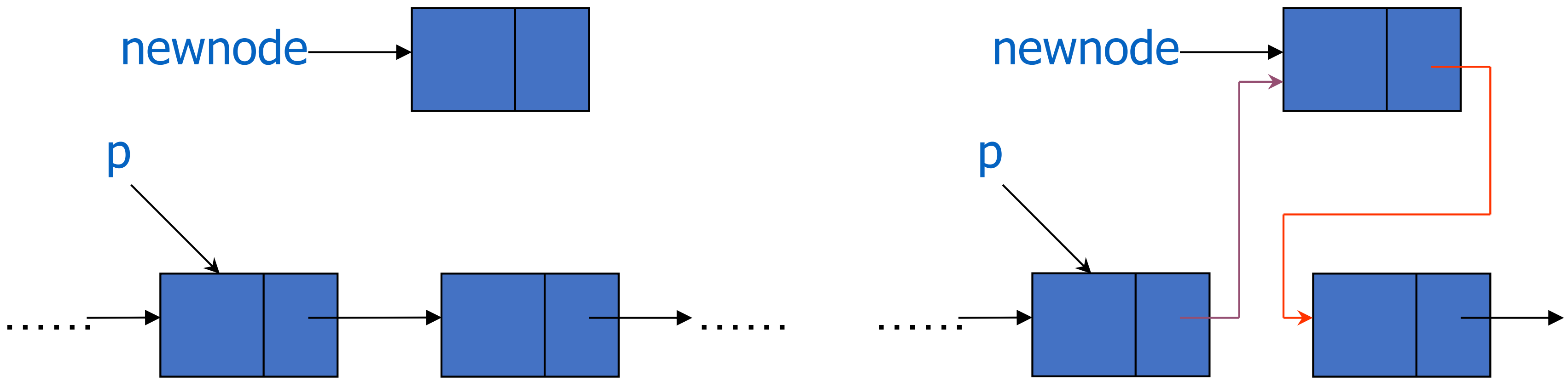
Review: Dummy Header Node

- 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



Review: Dummy Header Node

- One case remaining for insert



before insert

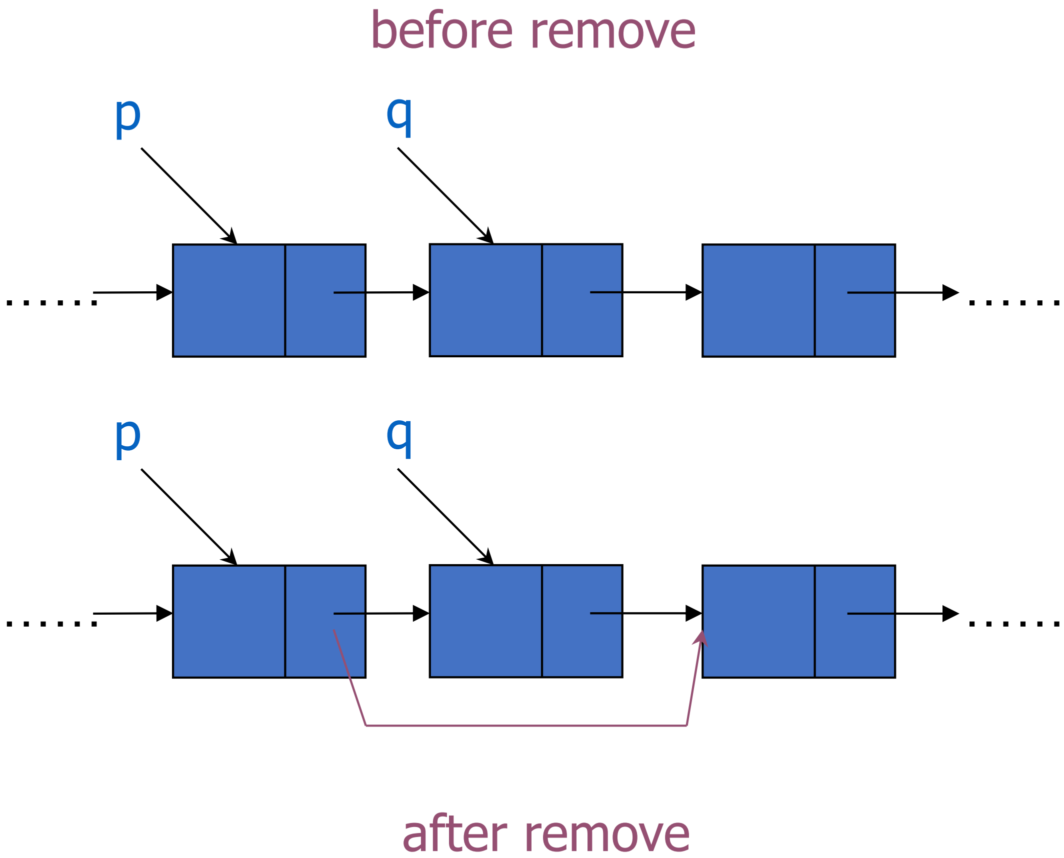
after insert

Review: Insert a node: With Dummy Header

```
Void List::InsertNode(ListNode* newnode)
{
        if (first==NULL)
                first=newnode;
        else if (newnode->data<first->data) {
                newnode->next=first;
                first=newnode;
        }
        else {
                ListNode* p=first;
                while(p->next!=NULL && newnode->data>p->next->data)
                        p=p->next;
                //p will stop at the last node or at the node
                //after which we should insert the new node
                //note that p->next might be NULL here
                newnode->next=p->next;
                p->next=newnode;
        }
}
```

Review: Dummy Header Node

- One case remaining for remove

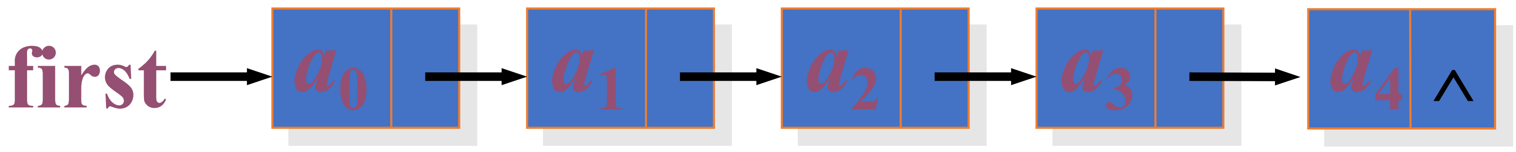


Review: Remove a node: With Dummy Header

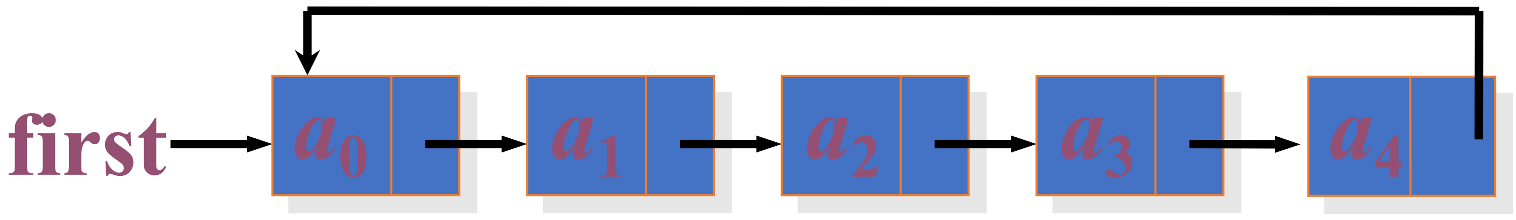
```
Void List::RemoveNode(ListNode* q)
{
        //remove at the beginning of a list
        if (q==first)
                first=first->next;
        else
        //remove not at the beginning
        {
                ListNode* p=first;
                while(p->next!=q)
                        p=p->next;
                p->next=q->next;
        }
}
```

Review: Circular Lists

- Suppose we are at the node a_4 and want to reach a_1, how?

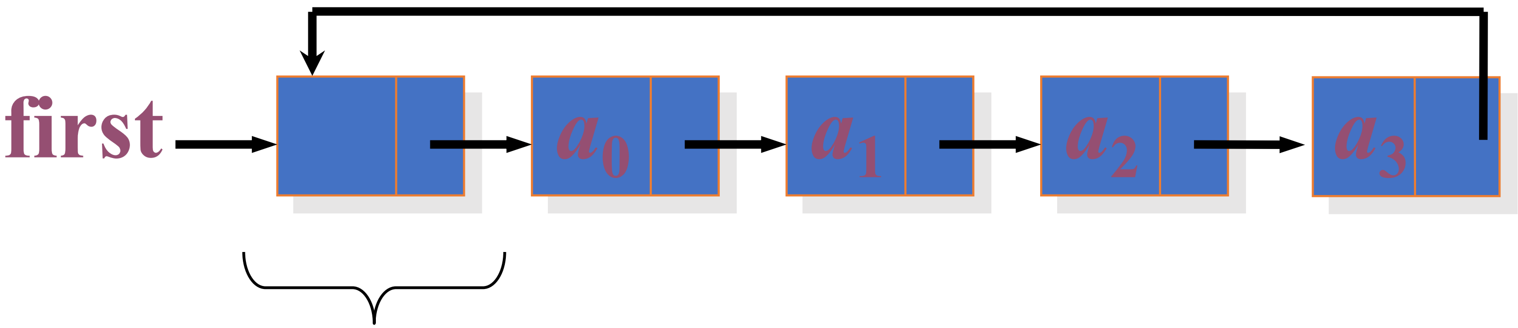


- If one extra link is allowed:



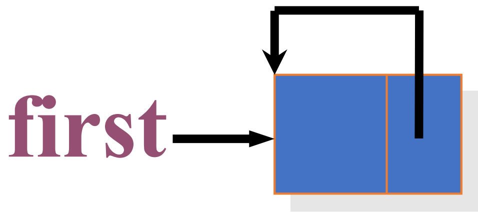
Review: Circular Lists

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



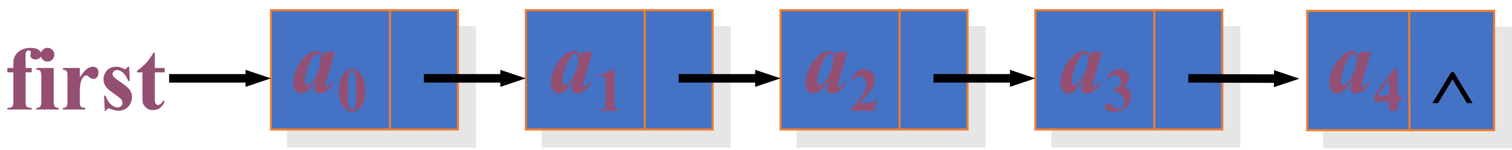
Dummy Header node

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

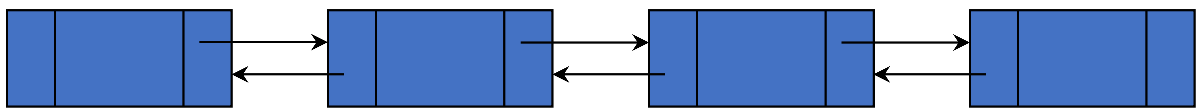


Review: Doubly Linked List

- Problem with singly linked list
  - When at a_4, we want to get a_3
  - When deleting node a_3, we need to know the address of node a_2
  - When at a_4, it is difficult to insert between a_3 and a_4

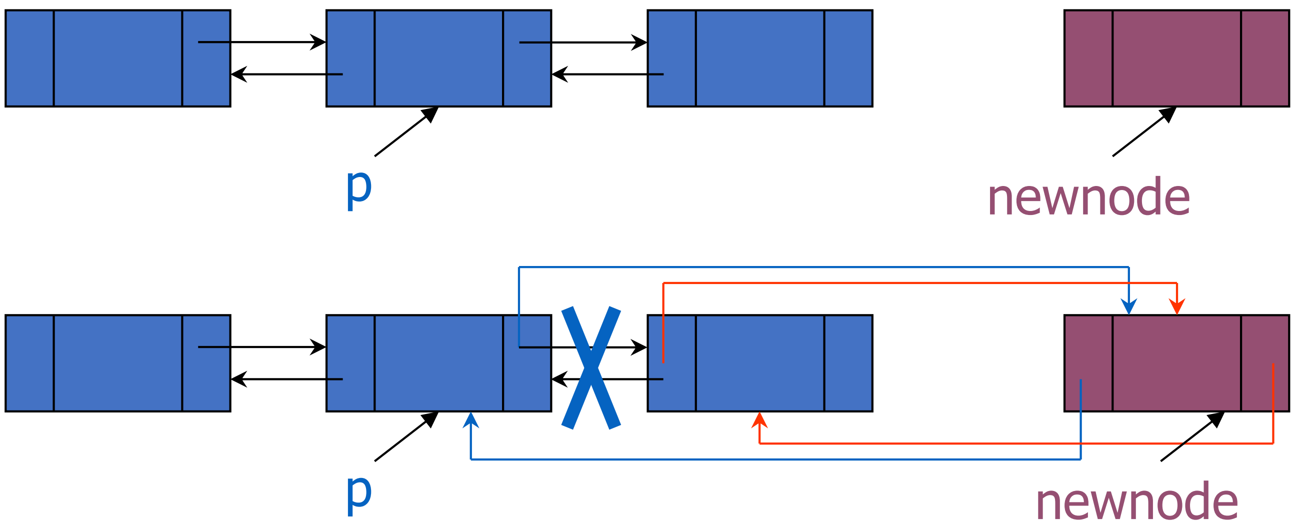


- If allowed to use more memory spaces, what to do?



Review: Doubly Linked List

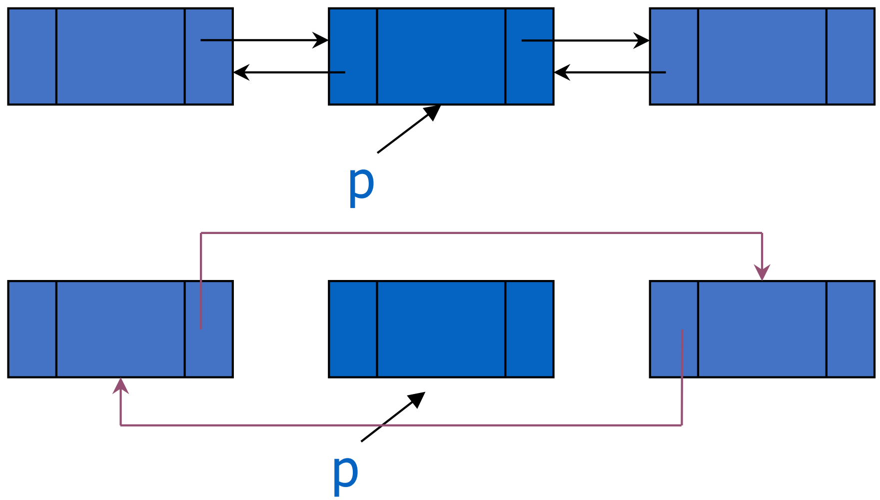
- To insert a node after an existing node pointed by p




```
DoublyLinkedList::InsertNode(ListNode *p, ListNode *newnode)
{
        newnode->Lnext=p;
        newnode->Rnext=p->Rnext;
        if(p->Rnext!=NULL) p->Rnext->Lnext=newnode;
        p->Rnext=newnode;
}
```

Review: Doubly Linked List

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




```
DoublyLinkedList::RemoveNode(ListNode *p)
{
        if(p->Lnext!=NULL) p->Lnext->Rnext=p->Rnext;
        if(p->Rnext!=NULL) p->Rnext->Lnext=p->Lnext;
}
```

Review: Advantages / Disadvantages of Linked List

Linked allocation: Stores data as individual units and link them by pointers.

Advantages of linked allocation:

- 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
- 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

Disadvantages:

- Take up additional memory space for the links
- Accessing random parts of the list is slow. (need to walk sequentially)

Exercise 1

Given a singly linked list, find the middle of the linked list.

- For example, if the given linked list is *1->2->3->4->5*, then the output should be 3.
- If there are even nodes, then there would be two middle nodes, we need to print the second middle element. For example, if the given linked list is *1->2->3->4->5->6*, then the output should be 4.

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

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

```
void List::printMiddle()
{
            . . .
}
```

Exercise 1

```
void List::printMiddle()
{
   if (first!=NULL)
   {
         int len = size();
          ListNode * temp = first;

         // traverse till we reached half of length
         int midIdx = len / 2;
         while (midIdx--)
            temp = temp->next;

         cout << "The middle element is " << temp->data << endl;
   }
}
```

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

```
class List
{
public:
        List( String  );
        List();
        int size();
private:
        ListNode *first;
        string name;
}
```

Exercise 2

Given a pointer to the head node of a linked list, the task is to reverse the linked list. We need to reverse the list by changing the links between nodes.

- E.g., Input: the following linked list *1->2->3->4->NULL*
  Output: Linked list should be changed to *4->3->2->1->NULL*

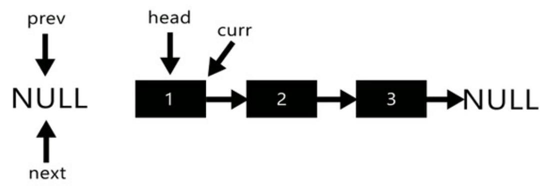
```
class ListNode
{
public:
      ListNode( int );
      ListNode( int, ListNode *);
      ListNode *get_Next()
      …
private:
      int data;
      ListNode *next;
};
```

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

```
void List::reverse()
{
              . . .
}
```

Exercise 2

```
void List::reverse ()
{
    // Initialize current, previous and next pointers
    Node* current = first;
    Node *prev = NULL, *next = NULL;

    while (current != NULL) {
      // Store next
      next = current->next;
      // Reverse current node's pointer
      current->next = prev;
      // Move pointers one position ahead.
      prev = current;
      current = next;
    }
    first = prev;
}
```




```
while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
```

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

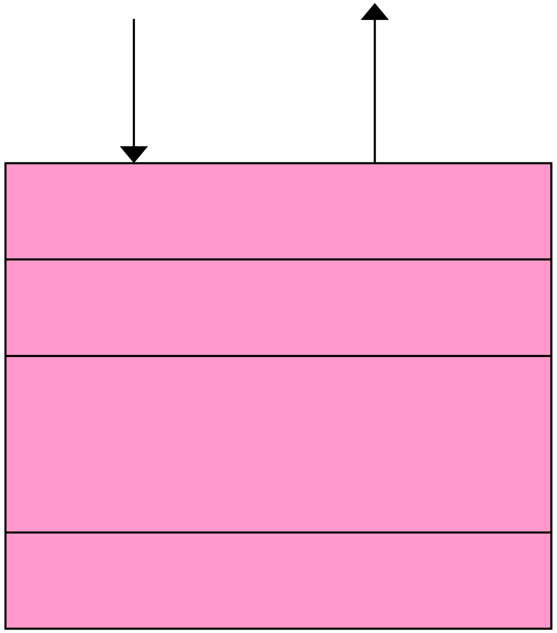
```
class List
{
public:
        List( String  );
        List();
        int size();
private:
        ListNode *first;
        string name;
}
```

Objective

- Stack Abstract Data Type
- Sequential Allocation
- Linked Allocation
- Applications

Stack

- Stack is a list with the restriction that insertions and deletions (usually all the accesses) can only be performed at one end of the list
- Also known as: Last-in-first-out (LIFO) list



ADT of Stack

Value:

A sequence of items that belong to some data type ITEM_TYPE

Operations for a stack *s*:

1. Boolean IsEmpty()
   Postcondition: If the stack is empty, return true, otherwise return false
2. Boolean IsFull()
   Postcondition: If the stack is full, return true, otherwise return false
3. ITEM_TYPE Pop() /*take away the top one and return its value*/
   Precondition: s is not empty
   Postcondition: The top item in s is removed from the sequence and returned
4. ITEM_TYPE top() /*return the top item's value*/
   Precondition: s is not empty
   Postcondition: The value of the top item in s is returned
5. Void Push(ITEM_TYPE e) /*add one item on top of the stack*/
   Precondition: s is <u>not full</u>
   Postcondition: e is added to the sequence as the top one

Array Implementation of Stack

```
// MyStack.h
#include “stdlib.h”
{
	public class MyStack
	{
		public:
				MyStack( int );
				bool IsEmpty();
				bool IsFull();
				void push(int );
				int pop();
				int top();
		private:
				int* data;
				int top;
				int MAXSize;
	};
}
```

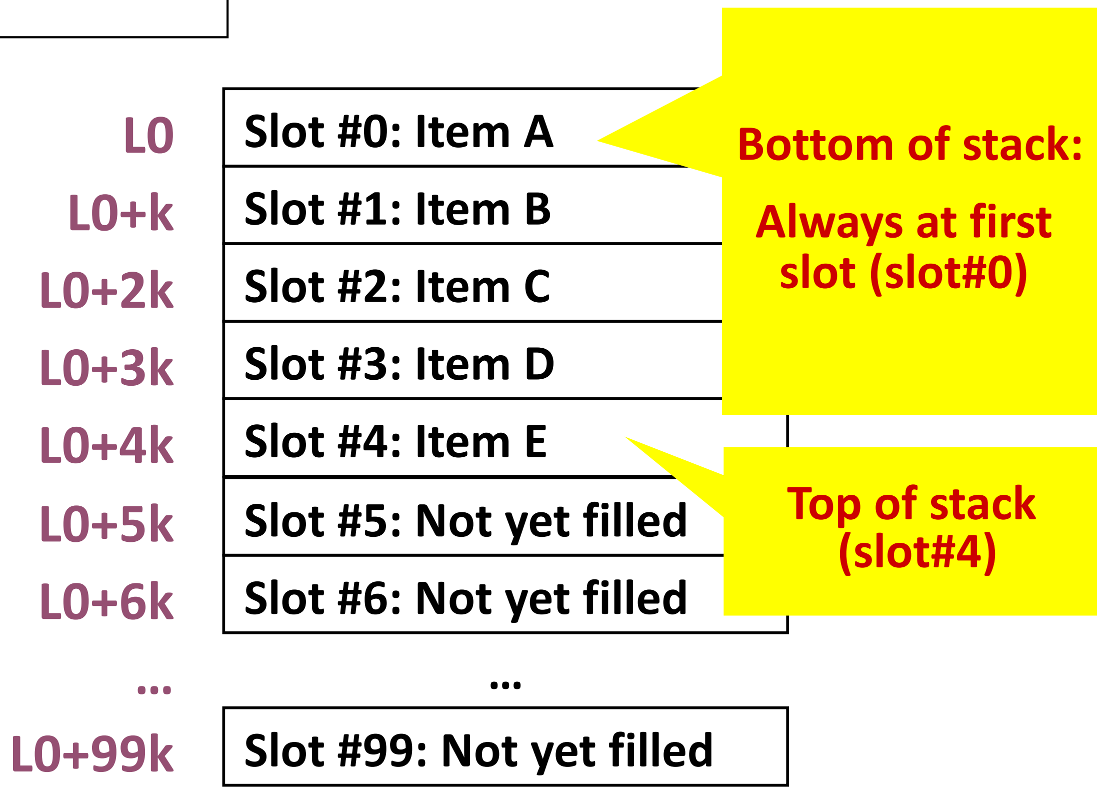
```
// MyStack.cpp

#include “MyStack.h”
MyStack::MyStack(int size)
{
			data=new int[size];
			top=-1;
			MAXSize=size;
}
bool MyStack::IsEmpty()
{
			return (top==-1);
}
bool MyStack::IsFull()
{
			return (top==MAXSize-1);
}
```

Array Implementation of Stack

```
// StackTest.cpp
#include "MyStack.h"
int main()
{
        MyStack* TS=new MyStack(100);
}
```

In computer memory, Suppose

- Size of each item is *k*.
- Base address is *L0*.



Array Implementation of Stack

When the stack is empty,

Top of stack is undefined

Slot #0: Not yet filled
Slot #1: Not yet filled
Slot #2: Not yet filled
Slot #3: Not yet filled
Slot #4: Not yet filled

…

Slot #99: Not yet filled

When the stack is *FULL*,

Slot #0: filled
Slot #1: filled
Slot #2: filled
Slot #3: filled
Slot #4: filled

…

Slot #99: filled

Top of stack is at slot #99 ie. Slot #(MAXSTACKSIZE-1)

Array Implementation of Stack: push

```
. . .
private:
    int* data;
    int top;
    int MAXSize;

void MyStack::push(int x)
{
    if (!IsFull() )
    {
        top=top+1;
        data[top] = x;
    }
    else
            ....
}
```

To "push" an item onto the stack

- Check whether not yet full.
- Increase the top indicator (slot number) of the stack.
- Copy the item to the top position immediately.

Slot #0: filled
Slot #1: filled
Slot #2: filled
Slot #3: to be filled
Slot #4: not yet filled
…
Slot #99: not yet filled

Top of stack: slot #2 => 3

Array Implementation of Stack: pop

```
. . .
private:
      int* data;
        int top;
        int MAXSize;


int MyStack::pop( )
{     int rtn_value;
      if (!IsEmpty())
      {
           rtn_value=data[top];
           top=top-1;
           return rtn_value;
      }
      else
      …
}
```

To "pop" an item from the stack (to take away the top one and return its value)

- Check whether it is empty.
- Save the value of item at the top position (to return it later)
- Decrease the top indicator (slot #)
- Return the saved value.
- *No need to clear any slot.*

Slot #0: filled
Slot #1: filled
Slot #2: filled
Slot #3: to be popped
Slot #4: not yet filled

…

Slot #99: not yet filled

Top of stack:
slot #3 => 2

Array Implementation of Stack: top

```
. . .
private:
      int* data;
        int top;
        int MAXSize;

int MyStack::top( )
{
      if (!IsEmpty())
      {
              return (data[top]);
      }
      else
      …
}
```

To return the value of an item from the stack (the top item)

- Check whether it is empty.
- Return the value of the item at the top position.

Slot #0: filled
Slot #1: filled
Slot #2: filled
Slot #3: to be returned
Slot #4: not yet filled
…
Slot #99: not yet filled

Top of stack: slot #3 (no change)

Exercise 1

Suppose an intermixed sequence of stack push and pop operations are performed. The pushes push into the stack the integers 0 through 9 in order; popped values are printed in the order they are popped.
Which of the below sequences ***could*** occur as the printed output?

a. 1 2 3 0 6 5 4 7 8 9
b. 2 3 4 5 6 7 8 9 0 1
c. 6 7 8 9 5 4 3 2 1 0
d. 7 8 9 6 5 4 2 3 1 0

Stacks: Use Dynamic Array

- How to choose the size of array data[]?
  - As we insert more and more, eventually the array will be full

- Solution: Use a dynamic array
  - Maintain capacity of data[]
  - Double capacity when size=capacity (i.e. full)
  - Half capacity when size $\leq$ capacity/4

- Question: What if we change capacity/4 to capacity/2 ?
  - E.g., initial cap is 4; **I** means insertion, **D** means deletion;
  - I, I, I, I, I (expand; cap=8, size=5), D (shrink; cap=4, size=4), I (expand; cap=8, size=5), D (shrink; cap=4, size=4), I (expand), D (shrink), ...

Stacks: Another implementation

```
class Stack
{
        public:
          Stack(int initCap=100);
          Stack(const Stack& rhs);
          ~Stack();

          void push(Item x);
          void pop(Item& x);

        private:
          void realloc(int newCap);
          Item* data;
          int size;
          int cap;
};
```

```
// An internal func. to support resizing of array
void Stack::realloc(int newCap) {
      if (newCap < size) return;
      //oldarray "point to" data
      Item *oldarray = data;

      //create new space for data with size newCap
      data = new Item[newCap];
      for (int i=0; i<size; i++)
            data[i] = oldarray[i];
      cap = newCap;
      delete [] oldarray;
}

void Stack::push(Item x) {
      if (size==cap)  realloc(2*cap);
      array[size++]=x;
}
```

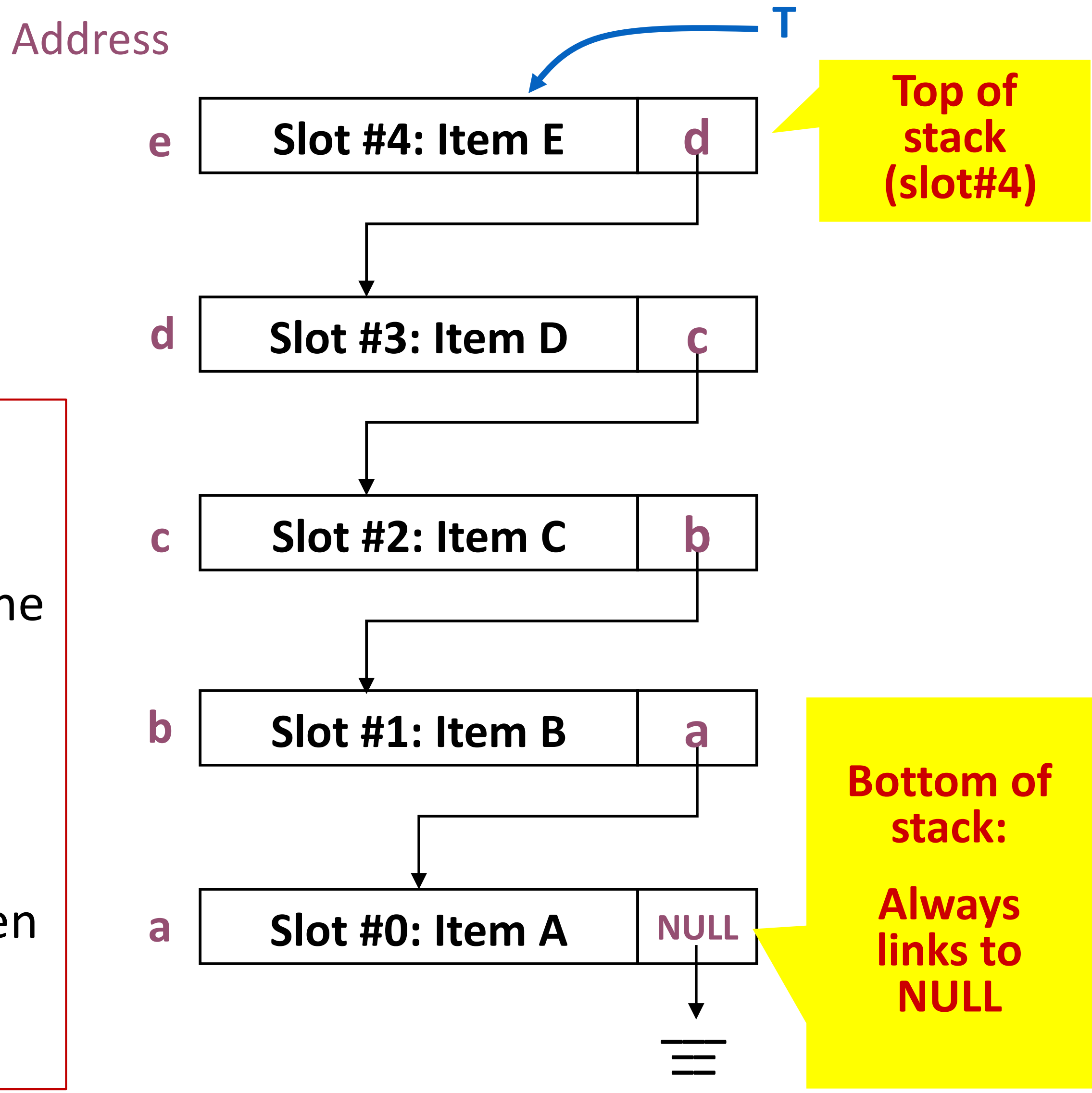
Stacks: Another implementation

```
void Stack::pop(Item& x)
{
     // assume EmptyStack is a special value
     if (size==0)
          x=EmptyStack;
     else
     {
          x=array[--size];
          if (size <= cap/4)
               realloc(cap/2);
     }
}
```

Linked Implementation of Stack

Stack can also be implemented with **linked list**.

- Typically, a pointer points to the top of the stack. (T)
- When the stack is empty, this pointer will be NULL.
- Each slot is allocated only when it is needed to store an item.



Linked Implementation of Stack

```
// MyStack.h


#include "stdlib.h"
#include "ListNode.h"
{
    class MyStack
    {
    public:
        MyStack( );
        Pop();
        IsEmpty();
        Push(int );
        …
    private:
        ListNode *Top;
    };
}
```

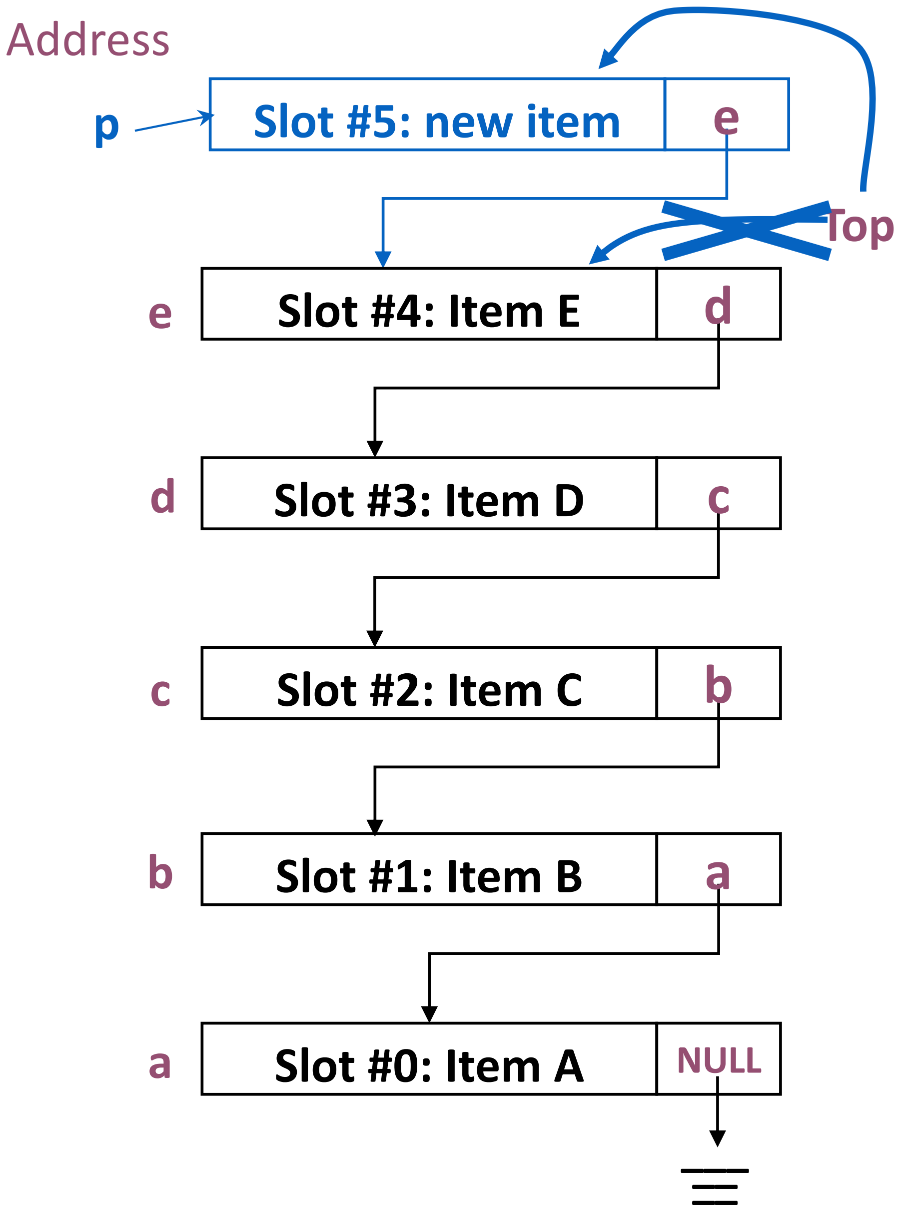
```
// ListNode.h

#include "stdlib.h"
{
    class ListNode
    {
    public:
        ListNode( int );
        ListNode( int, ListNode *);
        ListNode *get_Next()
        {
            return next;
        }
        …
    private:
        int data;
        ListNode *next;
    };
}
```

Linked Implementation of Stack: push

Push: To insert new information onto the top of the stack

- Allocate memory for an auxiliary pointer **p**
- Put new item into p->data
- p->next = T
- T=p

```
void MyStack::Push (int new_item)
{
    ListNode* p;
    p=new ListNode(new_item, Top);
    // p->data = new_item;
    // p->next = Top;
    Top = p;
}
```

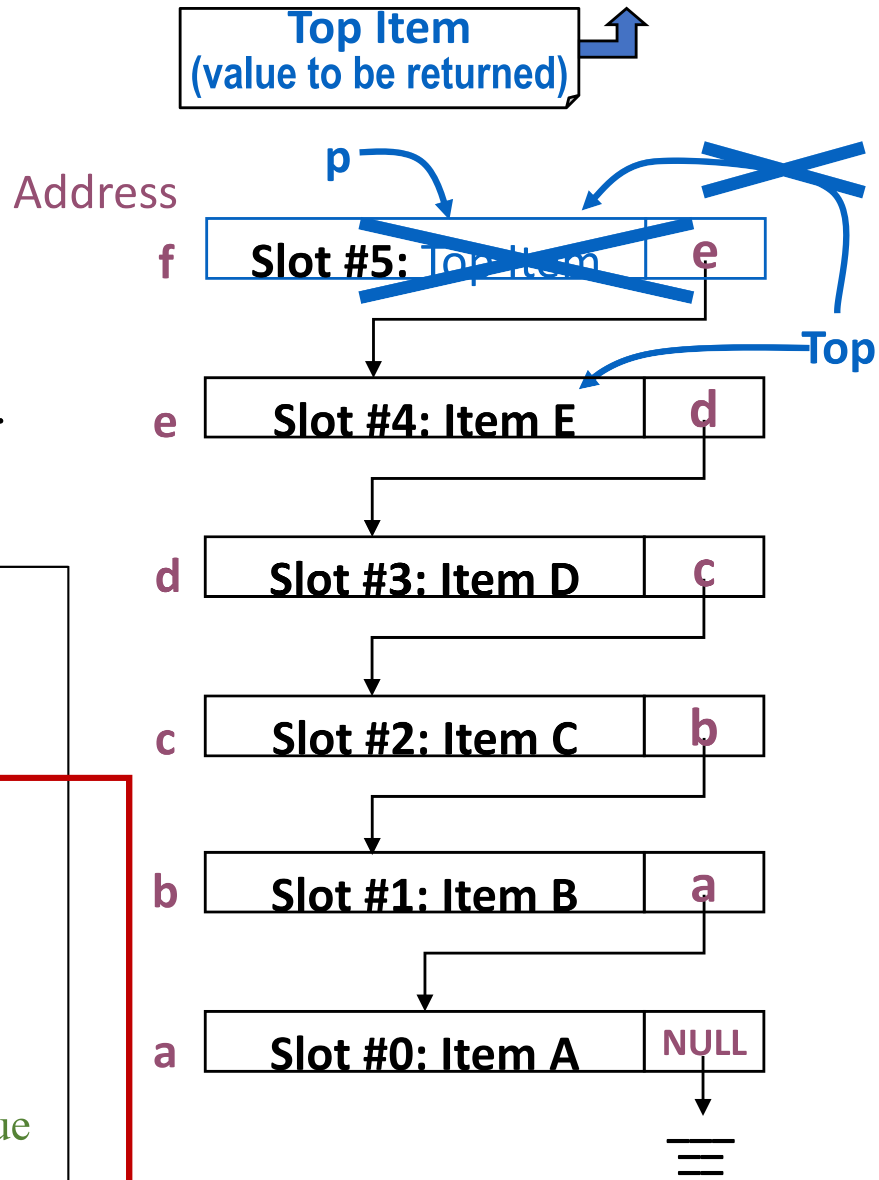


Linked Implementation of Stack: pop

Pop: To take away (and delete) the top item and return its value.

- Check whether the stack is empty.
- Store the value of the item so that we can return it later.
- Update the T pointer to point to the next item.
- Return the value of the top item.

```
int MyStack::Pop () {
ListNode* p;    //a pointer to point to original top node
int rtn_value;  //the value of the item to be returned
if (IsEmpty())  //check whether the stack is empty
{ //Exception handling }
rtn_value=Top->data;  //save the value to be returned
Top= Top->next;       //update the T pointer
return (rtn_value);   //return the original top node value
}
```



Linked Implementation of Stack: pop

```
int MyStack::Pop () {
        ListNode* p;    //a pointer to point to original top node
        int rtn_value;   //the value of the item to be returned
        if (IsEmpty())  //check whether the stack is empty
        { //Exception handling }
        rtn_value=Top->data;  //save the value to be returned
        ListNode* temp = Top;
        Top= Top->next;        //update the T pointer
        delete temp;
        return (rtn_value);      //return the original top node value
}
```

Exercise 1

find and remove the largest element in a stack.

```
// MyStack.h
#include "stdlib.h"
{
	public class MyStack
	{
		public:
				MyStack( int );
				bool IsEmpty();
				bool IsFull();
				void push(int );
				int pop();
				int top();
		private:
				int* data;
				int top;
				int MAXSize;
	};
}
```

```
int MyStack::find_and_Remove_max_val()
{
		. . .
}
```

Input: a stack with 5 2 7 4 (4 is at top)
Output: a stack with 5 2 4 and return 7

Exercise 1

find and remove the largest element in a stack.

```
int find_and_Remove_max_val()
{
    if (isEmpty()) {
        cout << "Stack is empty" << endl; return -1;
    }

    int maxElement = INT_MIN;
    Stack temp;
    while (!isEmpty()) {
        int element = pop();
        if (element > maxElement) {
            maxElement = element;
        }
        temp.push(element);
    }

    while (!temp.isEmpty()) {
        int element = temp.pop();
        if (element != maxElement)
            push(element);
        }
    return maxElement;
}
```

Exercise 2

Given string str, we need to print the reverse of individual words.

```
// MyStack.h
#include "stdlib.h"
{
    public class MyStack
    {
        public:
                MyStack( int size);
                bool IsEmpty();
                bool IsFull();
                void push(char);
                char pop();
                char top();
        private:
                char* data;
                int top;
                int MAXSize;
    };
}
```

```
void reverseWords (string str)
{
        stack st (str.length());
        . . .
}
```

Input: Hello World
Output: olleH dlroW

Exercise 2

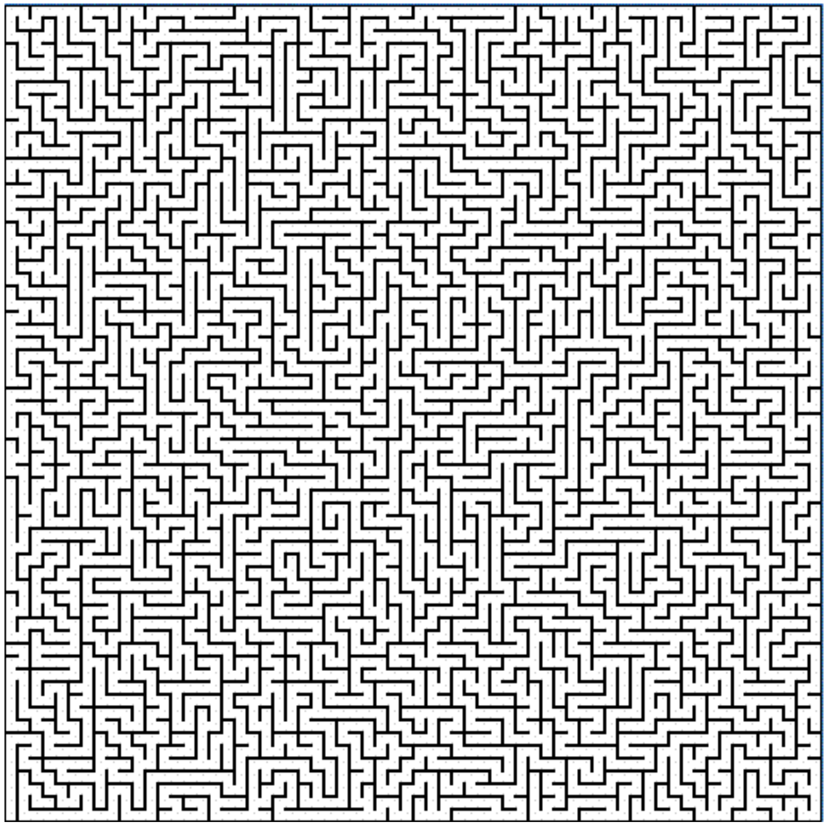
Given string str, we need to print the reverse of individual words.

```
void reverseWords (string str)
{
    stack st (str.length());

     for (int i = 0; i < str.length(); ++i)
    {
      if (str[i] != ' ')
          st.push(str[i]);
      else {
          while (st.empty() == false) {
              cout << st.pop();
          }
          cout << " ";
      }
    }

   //  there may not be space after last word
    while (st.empty() == false) {
       cout << st.pop();
     }
}
```

Application1: Backtracking

Generating a maze

Start (0, 0)



End (width-1, heigh-1)

Using stacks (simplest way)

1. Start from the entrance cell
2. Randomly select an unvisited neighbor cell of the stack top and break the wall, then push the new cell onto the stack
3. If all the neighbors are already visited, then go back by popping cells from the stack
4. Until the exit is reached

Try by yourself on a 4*4 maze!

Constructing a 4*4 maze

- Variables needed
  - An array memorizing whether a room is visited or not
  - A stack
  - An array memorizing whether a wall is broken or not
- How to solve a maze?

Application 2: Balancing Symbols

- When writing programs, we use
  - () parentheses [] brackets {} braces
- A lack of one symbol may cause the compiler to emit a hundred lines without identifying the real error
- Using stack to check the balance of symbols
  - [( )] is correct while [( ]) is incorrect

- Read the code until end of file
  - **If** the character is an opening symbol: ([{, **then** push it onto the stack
  - **If** the character is a closing symbol:)] }, **then** pop one (if the stack is not empty) from the stack to see whether it is the correct correspondence
  - Output error in other cases

Application 3 Evaluation of Postfix Expression

- Infix Expression Example: (A+B)*((C-D)*E+F)

  We need to add "(" and ")" in many cases.

- Postfix Expression Example: AB+CD-E*F+*

  Each operator follows the two operands.

  The order of the operators (left to right) determines the actual order of operations in evaluating the expression.

- Prefix expression Example : *+AB+*-CDEF

  Each operator precedes the two operands.

Application 3 Evaluation of Postfix Expression

6 2 + 3 1 - 4 * 7 + *

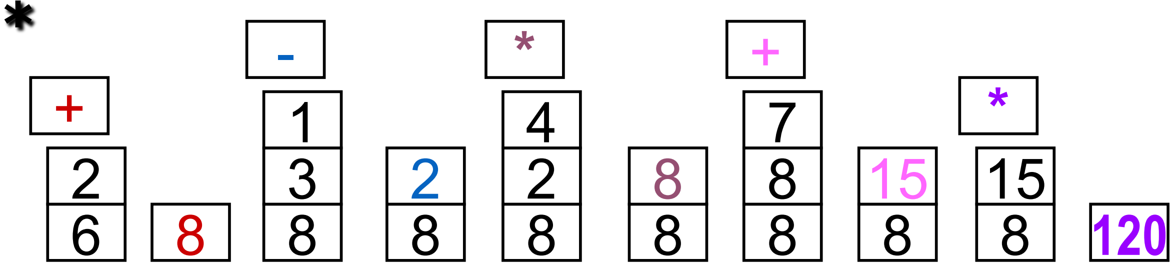
=8 3 1 - 4 * 7 + *

=8 2 4 * 7 + *

=8 8 7 + *

=8 15 *

=120



The method:

- Scan the expression from left to right.
- For each symbol, if it is an operand, we store them for later operation (LIFO) push
- If the symbol is an operator, take out the latest 2 operands stored and compute with the operator. pop pop

  Treat the operation result as a new operand and store it. push
- Finally, we can obtain the result as the only one operand stored. pop

Application 3 Evaluation of Postfix Expression

```
//check whether the parameter symbol is a digit
bool IsDigit(char symbol)
{
  if (symbol >= '0' && symbol <= '9’)          return true;
  return false;
}
int  Compute(char operator, int operand1, int operand2)
{
  switch (operator)
  {       case '+' :          return (operand1 + operand2);
          case '-' :          return (operand1 - operand2);
          case '*' :          return (operand1 * operand2);
          case '/' :          return (operand1 / operand2);
  }
}
```

Application 3 Evaluation of Postfix Expression

```
…
using namespace MyStack
BOOL IsDigit(char symbol) { .. }
int  Compute(char operator, int operand1, int operand2) { .. }
void main()
{
        int i, operand1, operand2, computed_value
        String * exp;
        wchar_t  c;
        //Input of expression: exp
        Console::Write(S"Enter the expression (no space in-between): ");
        exp=Console::ReadLine();
         //Compute the expression
        Stack *S=new Stack();
        for (i=0; i<exp->GetLength(); i++)
        {
           c=exp->get_Chars(i);
           if (IsDigit(c))
                S.push( (c-'0'));
                else
                {        operand2=S.pop();
                         operand1=S.pop();
                         computed_value=Compute(c,operand1,operand2);
                         S.push(computed_value);
                }
        }
        //Output the answer
        Console::Write(S"Answer: {0}", S.pop());
}
```

Application 4 Infix expression->postfix expression

Define the precedence relation
of some of the operators:

is the special symbol to denote the bottom of stack.

Operators	priority no.
#	0
(	1
+ or -	2
* or /	3

Example:(1+3)*((2-4)+5*7) => 1 3 + 2 4 - 5 7 * + *

Application 4 Infix expression->postfix expression

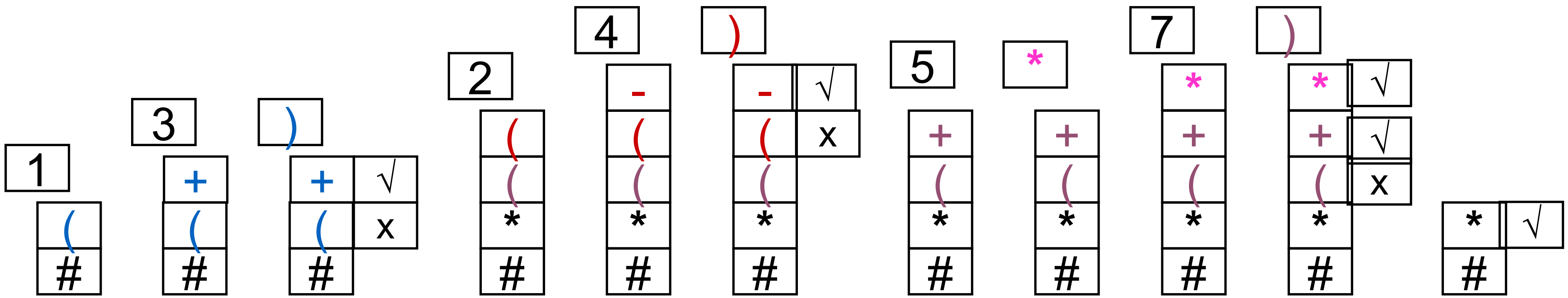
Define the precedence relation
of some of the operators:

is the special symbol to denote the bottom of stack.

Operators	priority no.
#	0
(	1
+ or -	2
* or /	3

Example:(1+3)*((2-4)+5*7) => 1 3 + 2 4 - 5 7 * + *

1	3	+	2	4	-	5	7	*	+	*



Learning Objectives

1. Explain the concepts of Stack
2. Understand the three functions of Stack
3. Able to use the three functions to generate and solve a maze
4. Fully understand how stack is used in Application 2

D:1; C:1,2; B:1,2,3; A:1,2,3,4