# CS3334 Data Structures
# Lec-1 Introduction & Linked Lists

## Lecturers & Instructors

- Lecture Instructor:
  - Dr.  Junqiao QIU,  Assist. Prof. at CS, YEUNG Y7708
  - Email: junqiqiu@cityu.edu.hk
  - Office Hours: W 15:30 - 16:30 or by appointment

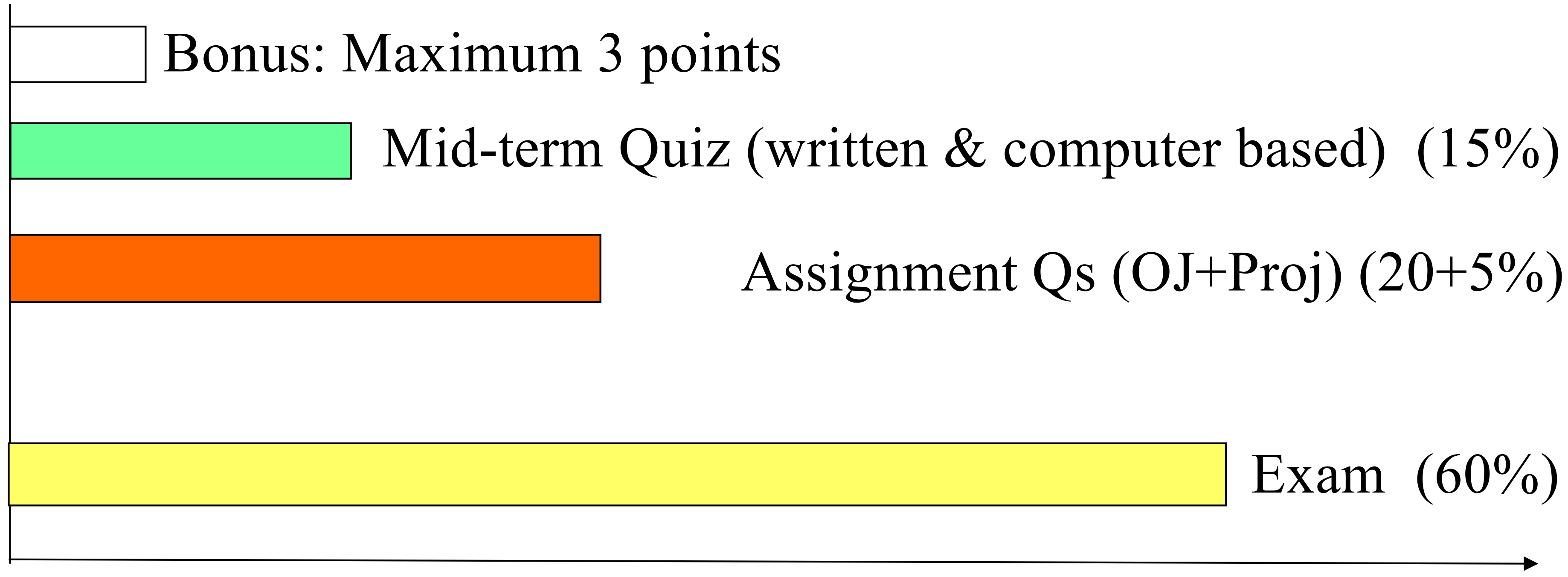
- Lab/Tutorial Instructor:
  - W13:00 - 13:50 QIU Junqiao
  - T17:00 - 17:50 CHEN Zixi
  - M18:00 - 18:50 HUANG Jiacheng

- Post your questions on Canvas for *quick response*
  - Canvas => Discussions

## Reference Books

- Cormen T., Leiserson C., Rivest R. and Stein C. (2009). *Introduction to Algorithms.* MIT Press, 3rd edition
- *Weiss M. Data Structures & Algorithm Analysis in C++. 4th Ed. (2013).*

## Assessment Pattern



**Compulsory: coursework >= 40**
**Compulsory: exam mark >= 30**

## Course Web Page

What you can find at the canvas system:

- Lecture Slides
  - ➢ Ready at least 24 hours before the lecture
- Lab/Tutorial Exercises
  - ➢ Selected questions are to be done before the  tutorials
  - ➢ Answers will be explained in tutorials and then program exercises will be practiced in and after the tutorial
- Assignments and solutions
- Announcement from the teacher
  - ➢ Important ones will also be distributed through email
- Q&A

## How to learn this course?

- Better not read the textbook ahead of time
- Think actively in lectures and tutorials
- Read the materials carefully after the lecture
- Do the assignments on your own
  - You may discuss with each other, study materials available on internet or refer to any book
  - **But the details should be entirely your work**
- Respond in time if you have any suggestions to the course or problems with the course

## Outcome Based T&L

- Any of you can get ***A*** if you work hard and achieve the outcomes
- Intended Learning Outcomes (ILO) will be shown at the end of every topic
- **Exams**: Mainly theory
- **Assignment and Quiz**: Programming
- Peer study (a large pool of potential exercises)

## Assessment Schedule

Current Lectures and Labs Schedule

(Tutorials: Week 3, 4, 5, 6, 7, 10, 11, 12)

- Week 1: Introduction & Abstract Data Type & Linked List
- Week 2: Linked List & Stack
- Week 3: Program Complexities, Lab
- Week 4: Queue & Hash, Lab
- Week 5: Tree & Heap & Game Tree, Lab
- Week 6: Balanced Binary Tree, review, Lab
- Week 7: Mid-term (Written), Lab
- Week 8: Mid-term (programming)
- Week 9: Disjoint Set, Suffix Array
- Week 10: Graphs, Lab
- Week 11: Sorting-1, Lab
- Week 12: Sorting-2, Concurrent Data Structures, Lab
- Week 13: Other Data Structures, review,

## Assignment questions

- Total 16 questions (20%)
- The last minute is the worst minute!
- Mapping number of questions x in the assignment to the assignment score

  $f(x) = 4 + 7x$ if $x<8$

  $f(x) = 12 + 6x$ if $8<=x<12$

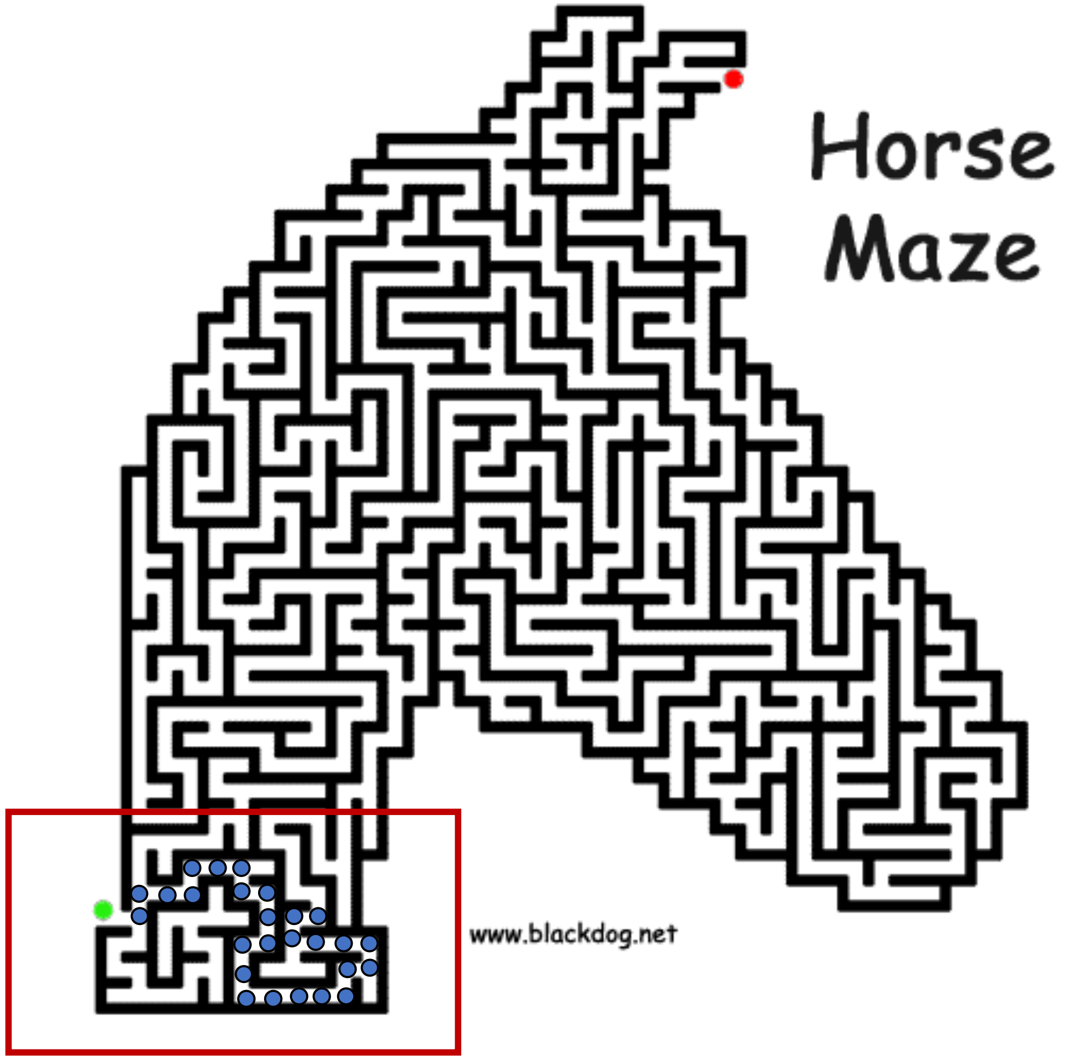
  $f(x) = 36 + 4x$ if $12<=x<=16$

- One Project (to be released in around Week 9)
- http://acm.cs.cityu.edu.hk/oj2/
- **must pursue your studies with academic honesty**

## Why do we need to learn Data Structures

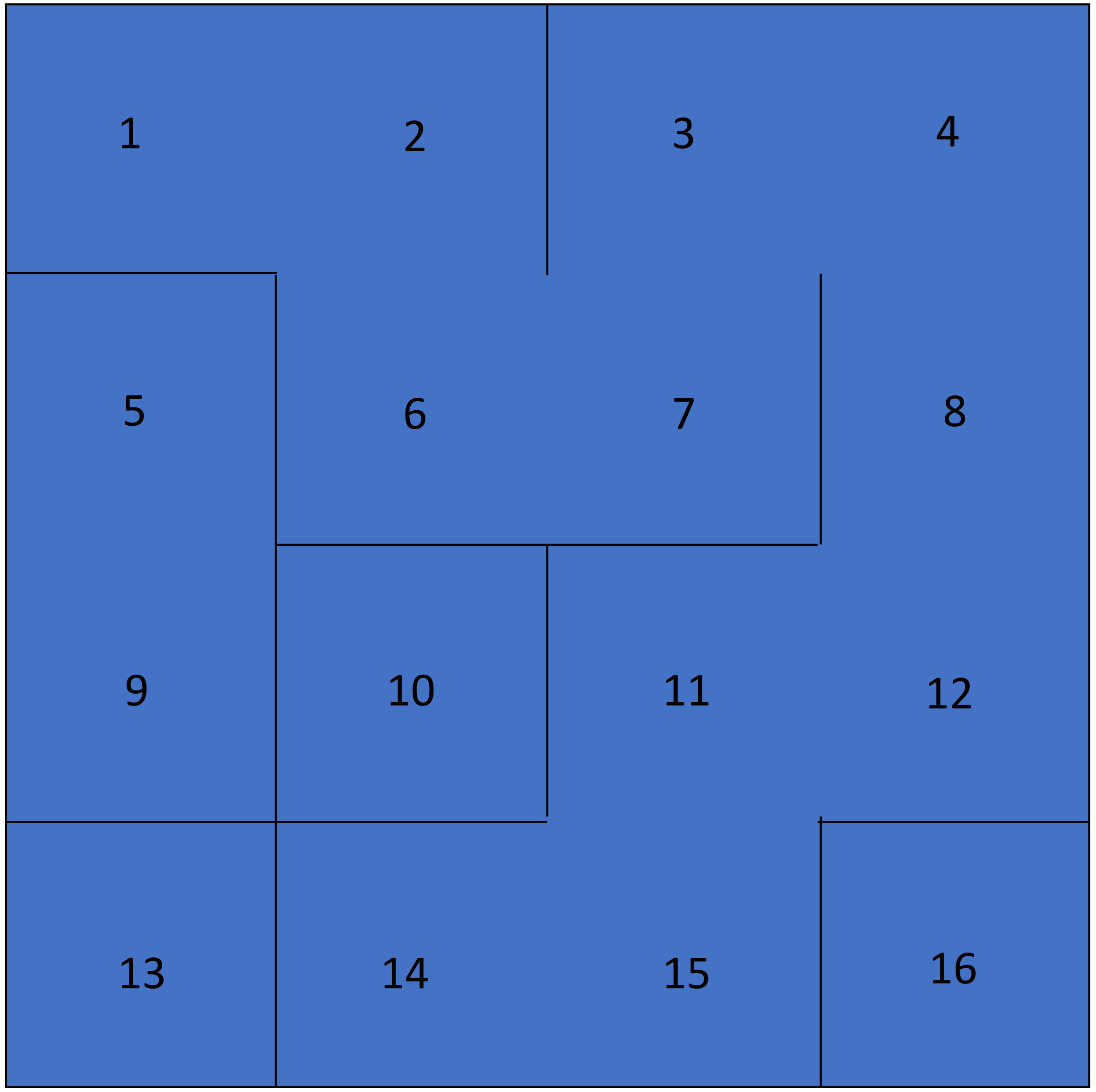
- One focus of Computer Science: Use suitable structures to save time and space
  - ➢1 second vs 1 year
  - ➢1KB vs 1GB
- When we design or play games
  - ➢How to design correctly?
  - ➢How to make the computer smarter?
  - ➢How to get information faster?
  - ➢How to reduce the game size?

## Good maze?



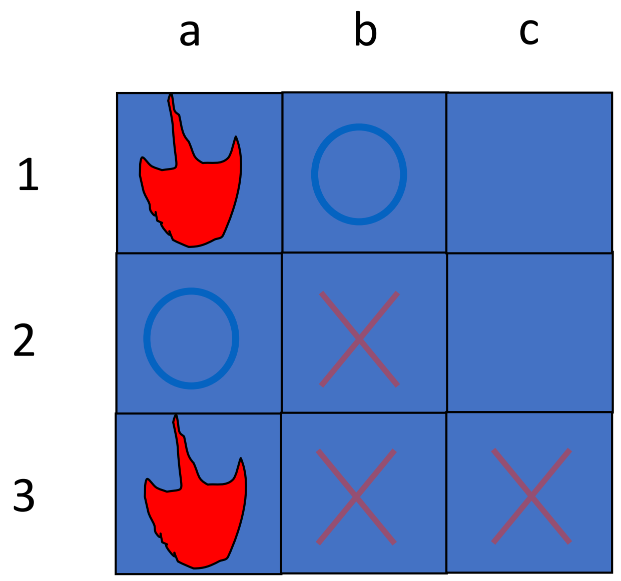
How to generate and solve mazes?

# Generating a Maze



- Union(1,2)
- Union(4,8)
- Union(6,7)
- Union(14,15)
- Union(5,9)
- Union(3,7)
- Union(2,6)
- Union(8,12)
- Union(3,4)
- Union(11,12)
- Union(11,15)
- Union(15,16)

## How to play smartly?



■ How should the computer compete with humans in turn-based games (like tic-tac-toe)?

# Does the player exist?

- Given the following list of numbers (player IDs), how to know 60 is in the list?

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 9 | 16 | 50 | 2 | 10 | 4 | 8 | 12 | 60 | 100 |

- If the list is sorted, how?

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 8 | 9 | 10 | 12 | 16 | 50 | 60 | 100 |

- Binary Search: Every round throw away half the numbers

## The ways to speed up

**1,000,000 players in the system**

**How to find Player information quickly given the player ID?**

Given PlayerID 5543098:

- **Method 1**: Check entry one by one?

  10000 seconds = ~ 3 hours

- **Method 2**: Sort and Binary Search?

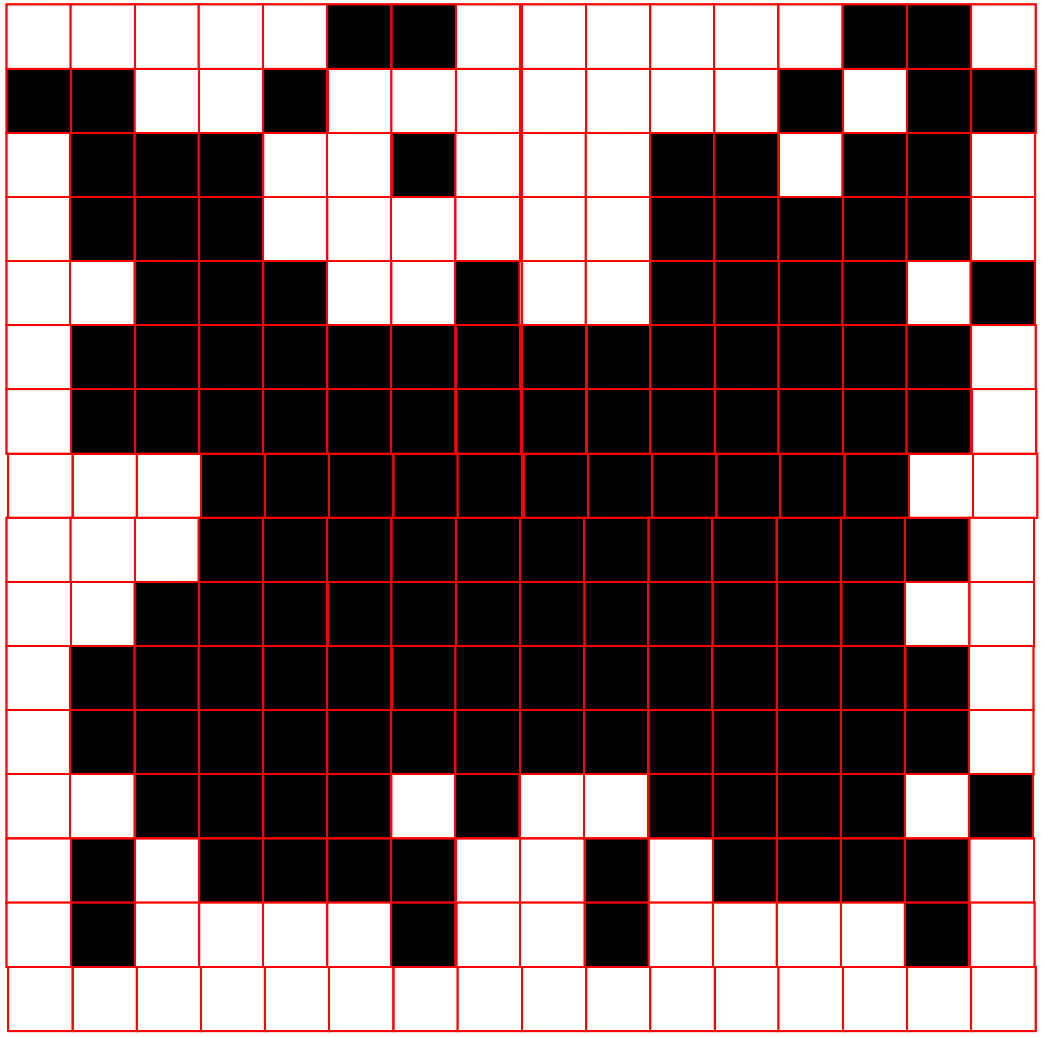
  1 second

- **Method 3**: Hash!

  Possibly 1/50 second

| **PlayerID** | **PlayerInfo** |
|---|---|
| 1453656 | … |
| 9876475 | … |
| 8769432 | … |
| …… | … |
| 5543098 | … |

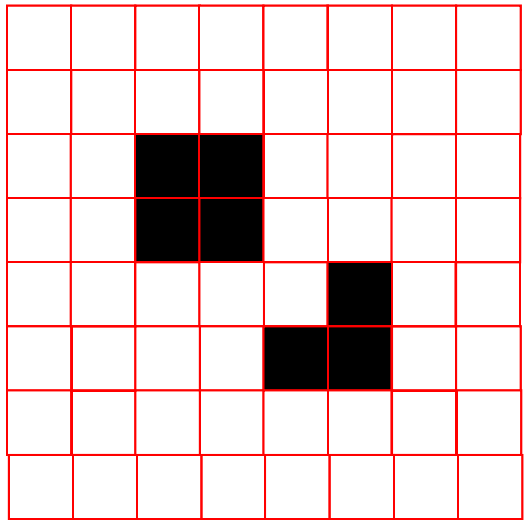
## How to store this figure?

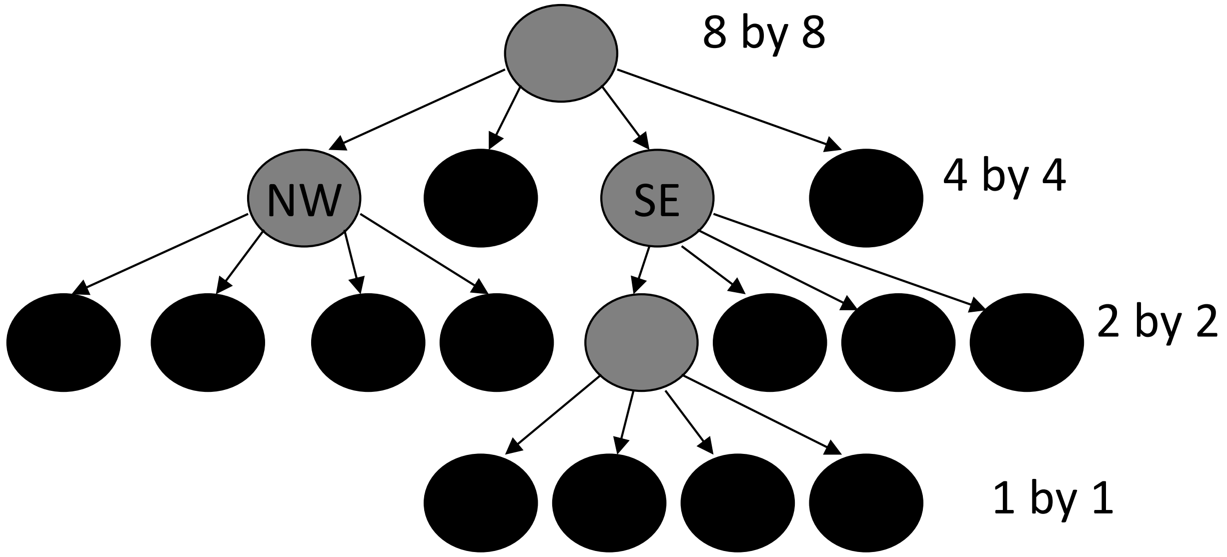


16 * 16

## How to save storage for a binary figure?

- Using quadtree to save space





- Keep decomposing the space by 4 until the colors within the same region are the same

## Outline about Linked Lists

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

## 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 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();
…
};
```

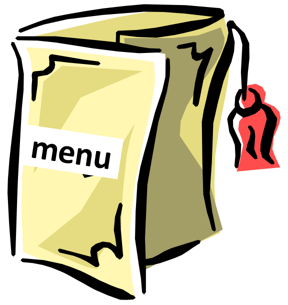
Some advanced data types are very useful.

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.



## 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.

## 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*

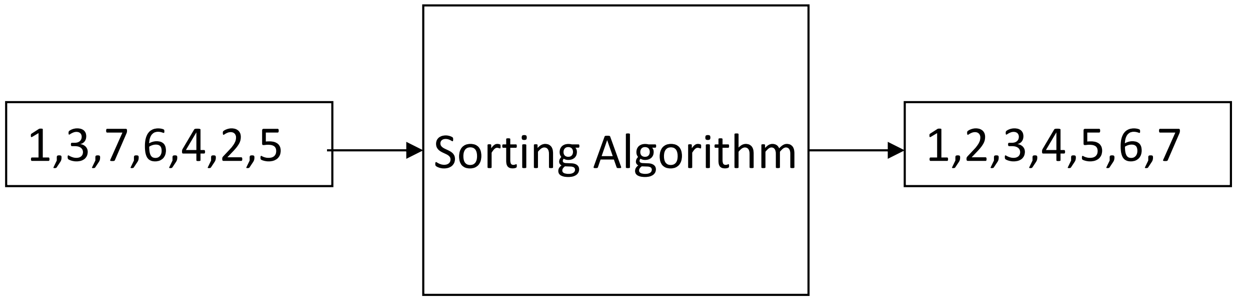
**3. int Size( )**

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

…

## How to solve a problem?

- Algorithms
  - a sequence of finite instructions
  - Given an input, can generate correct output
- Example:
  - Sorting



# Data Abstraction

Data abstraction is an important method in program design

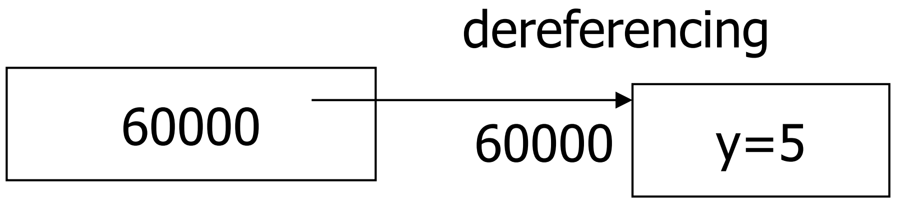
- Using the right data types => more efficient program
- Algorithms can be designed in terms of the abstract data type.
- We can defer decisions of data types until their uses are fully understood.

| Phase | Description | Result |
|---|---|---|
| 1. Problem Formulation | | A rough algorithm |
| 2. Design the algorithm (Suppose no ADT available yet.) | Construct a mathematical model + its operations (**ADT**)<br><br>e.g., the model is a *set*, the problem is to list all elements. This phase is to express the algorithm in pseudocode, while figuring out the picture of the ADT.<br><br>Implementation-independent:<br>Not to think whether to use array or linked list etc.. Not to design how to implement the operations. | The ADT<br><br>Pseudocode for algorithms |
| 3. Implement the algorithm | Implement the actual data types and the operations for the ADT | Data types and operations |

## Pointers and References

- Value Type: variable contains data of that type
  - int A; double B; …
- Reference Type: variable contains the address
  - String * c; Object * d; …
- Pointers

```
int y=5;
int *ypointer;
ypointer=&y;
*ypointer=?
```



- Exercise

```
int count = 1;
int &cRef = count;
int *cp = &count;
++cRef;
cp++;
```

What is the value of count?

## Call methods using pointers

```
GoodStudent *Alice=new GoodStudent();
If ( Alice->Knows(“Bob”) )
    cout<<“Alice knows Bob”;
If ( (*Alice).Knows(“Bob”) )
    cout<<“Alice knows Bob”;
```

- Remember: -> for pointer; . for dereference

## 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

## Passing Arguments

**Question:**

**When to use Call-by-value?**
**When to use Call-by-reference?**

For security of data, we apply Call-by-reference only if necessary, e.g.

(1) we really want the called function to change the data

(2) the data is too big, i.e.
Instead of copying a bulky data, e.g. a 100 byte record, (Call-by-value)
we prefer to copy the memory address only (Call-by-reference)

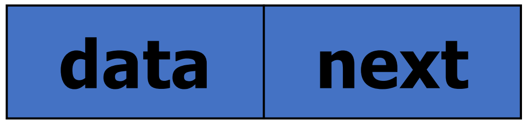
## List

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

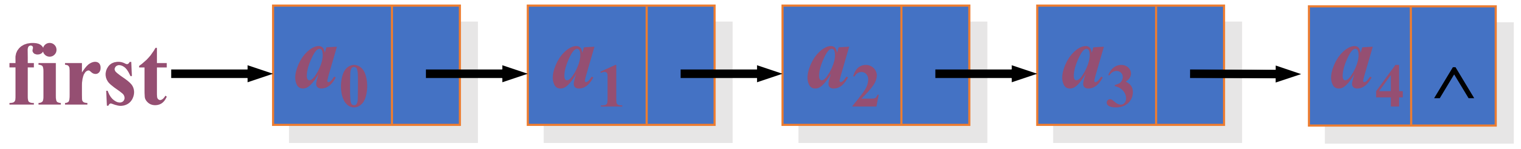
- is a sequence of *n* nodes $\{x_1, x_2, .., x_n\}$ whose essential structural properties involve only the relative positions between items as they appear in a line.
- can be implemented as
  - Arrays: statically allocated or dynamically allocated
  - Linked Lists: dynamically allocated
- A list can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

## Singly Linked List

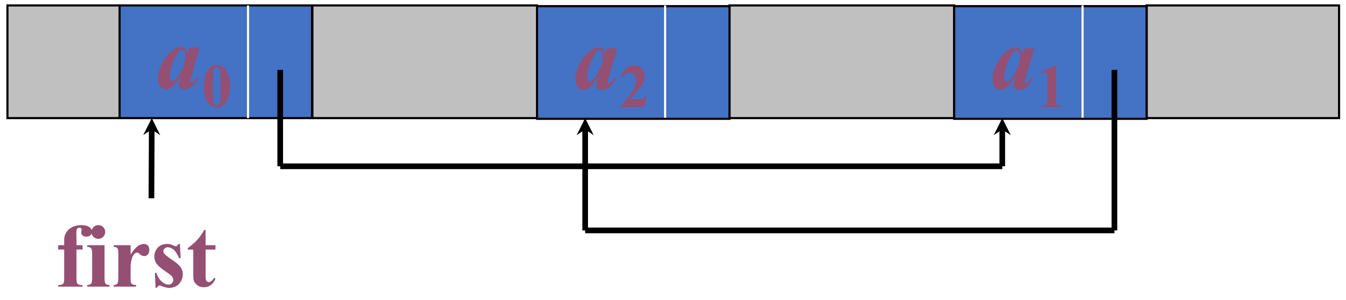
- Each item in the list is a node



- Linear Structure



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



## 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;
}
```

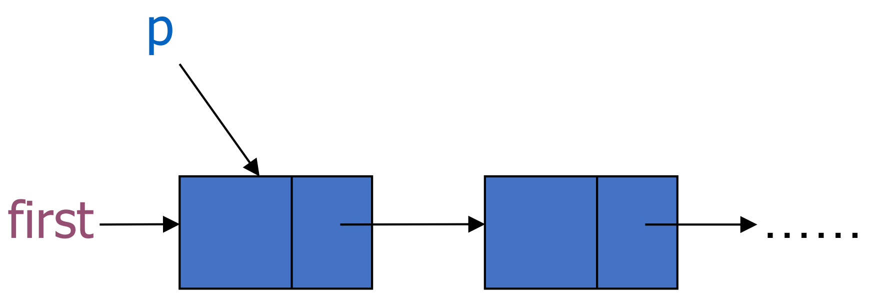
## Singly Linked List

- 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…

## Count a linked list

Use a pointer p to traverse the list

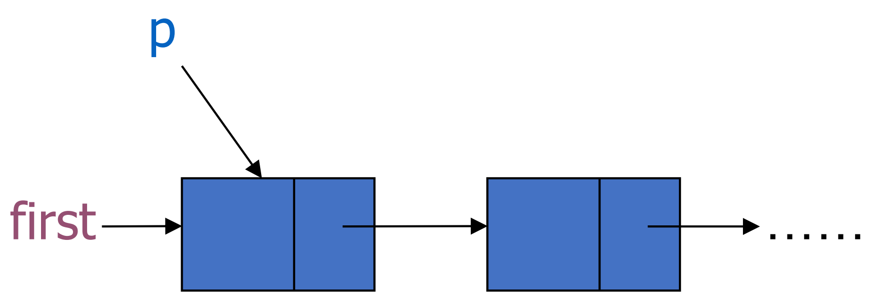
```
//List.cpp
int List::Count()
{
        int result=0;
        ListNode *p=first;
        while (p!=NULL)
        {
                result++;
                p= p->getNext();
        }
        return result;
}
```



# Print a linked list

Use a pointer p to traverse the list

```
//List.cpp
List::Print()
{
    ListNode *p=first;
    while (p!=NULL)
    {
        cout<<p->getData();
        p=p->getNext();
    }
}
```

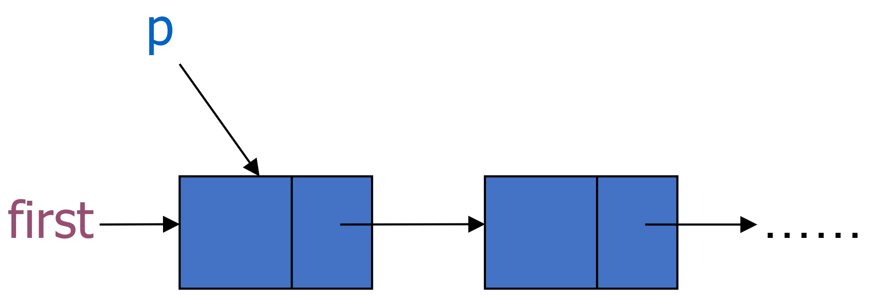


## Search for a node

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;
}
```



# Insert a node in Singly Linked List

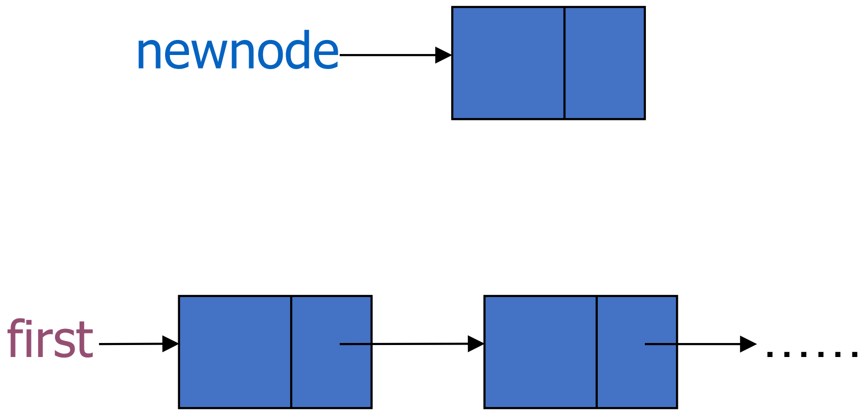
- Data come one by one, and we want to keep them sorted, how?
  - Do search to locate the position
  - Insert the new node

- 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

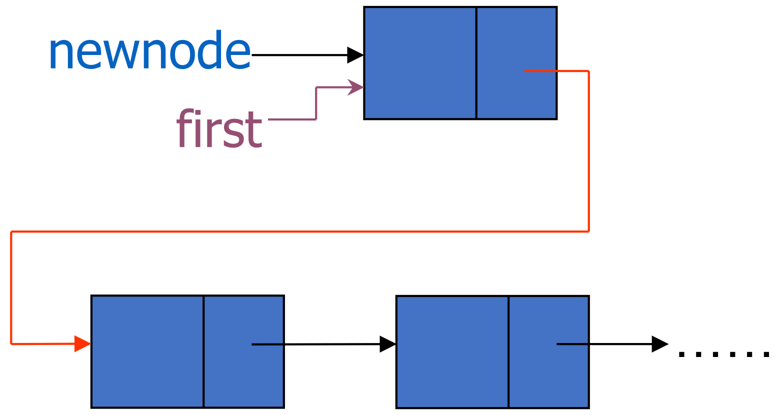
```
Void List::InsertNode(ListNode* newnode)
{
        if (first==NULL)
                first=newnode;
        else if (newnode->getData() < first->getData() )
                …
}
```

# Insert a node in Singly Linked List: Case 1

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



before insert



after insert

## Insert a node in Singly Linked List: Case 1

In the following slides, we assume all the class member variables are public, so that the writings will be more clear

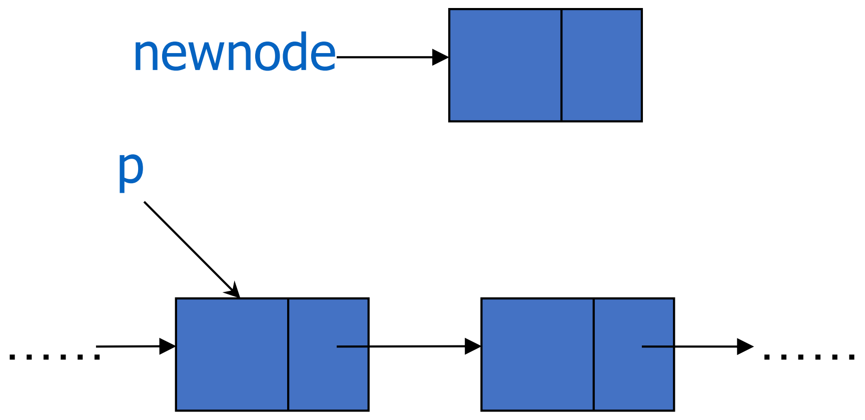
Remember that in real coding, you should use functions

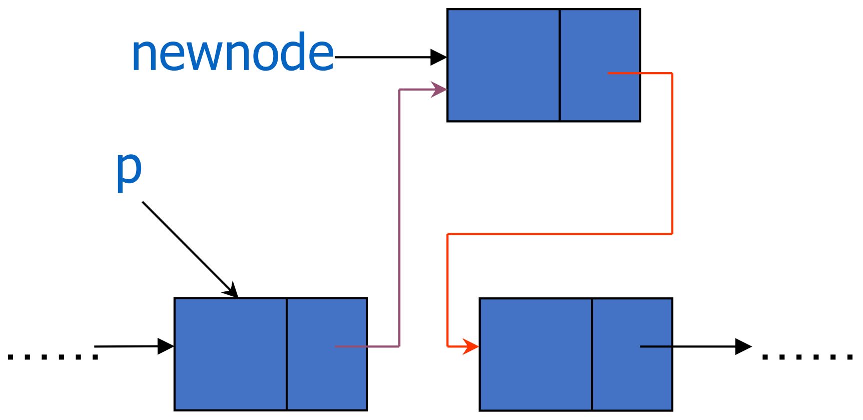
getData(), setData(), getNext(), setNext() to access private class variables

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

# Insert a node in Singly Linked List: Case 2

- Insert in the middle
  - newnode->next = p->next
  - p->next = newnode



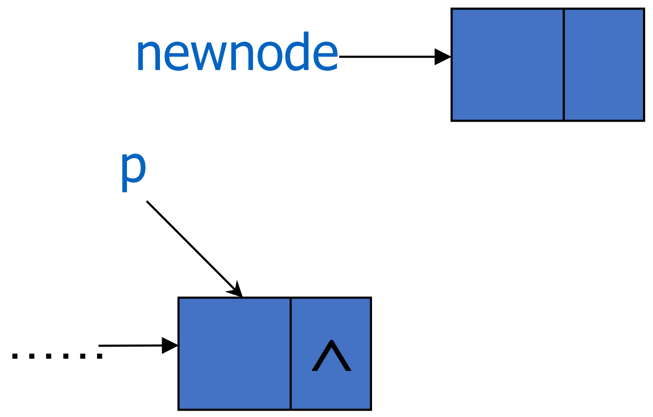


before insert

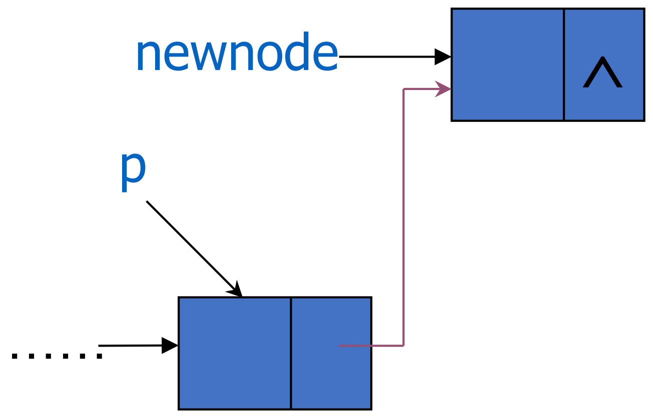
after insert

## Insert a node in Singly Linked List: Case 3

- Insert at the end of the list
  - newnode->next = NULL
  - p->next = newnode



before insert



after insert

## Insert a node: Complete Solution

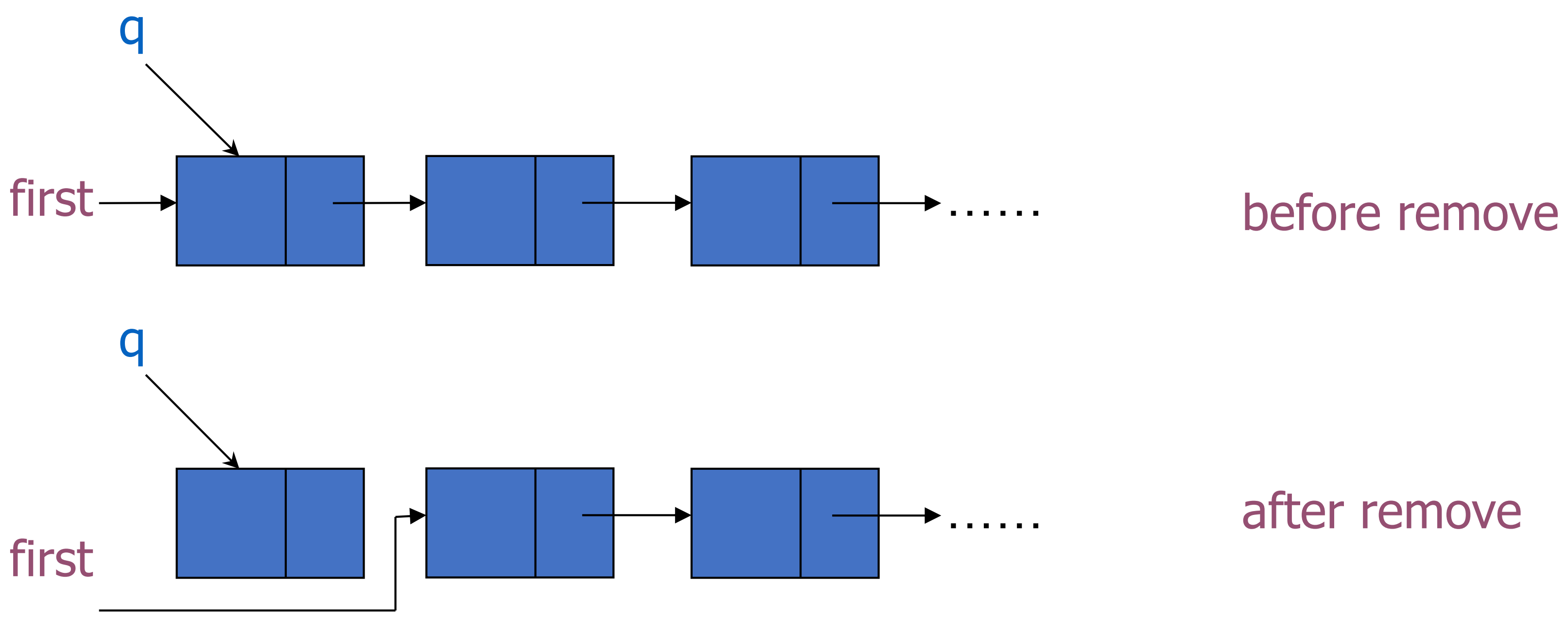
```
Void List::InsertNode(ListNode* newnode)
{
        if (first == NULL)
                first = newnode;
        else if (newnode->data < first->data) {
                newnode->next = first;
                first = newnode;
        }
        else {
                ListNode* p=first;
             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

## 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 “first” is unwanted
    - ➢The action we need: q=first; first=q->next

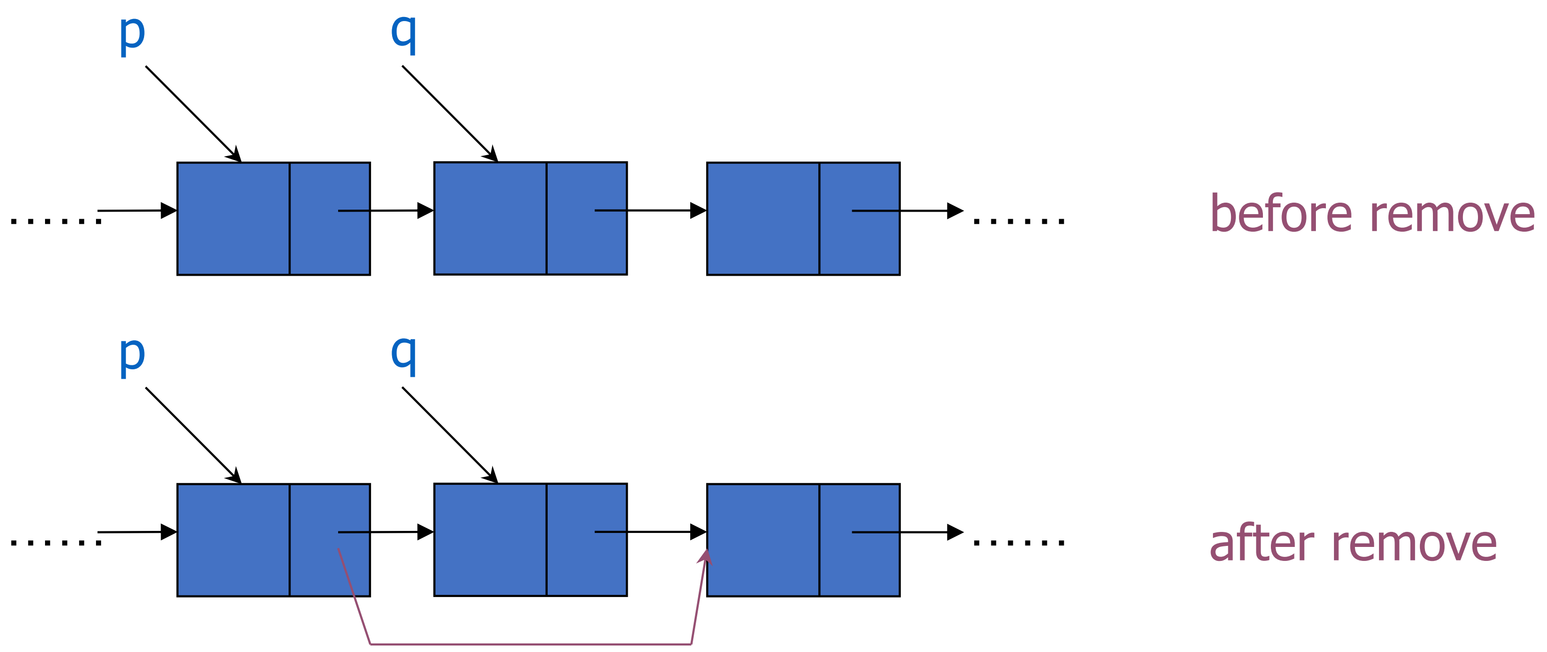


## Remove a node: Case 2

- 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



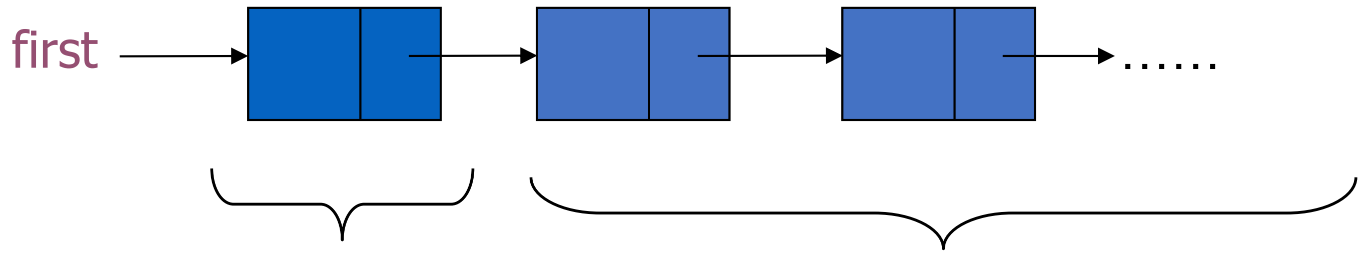
## Remove a node: Complete Solution

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

1. In real coding, you should also consider exceptions
2. Design of Linked List ADT is up to you, you can choose suitable functions to be member functions, e.g. InsertAtFront(), RemoveAtFront(). For each function, you can have your own way to implement it

## 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



Dummy Header node

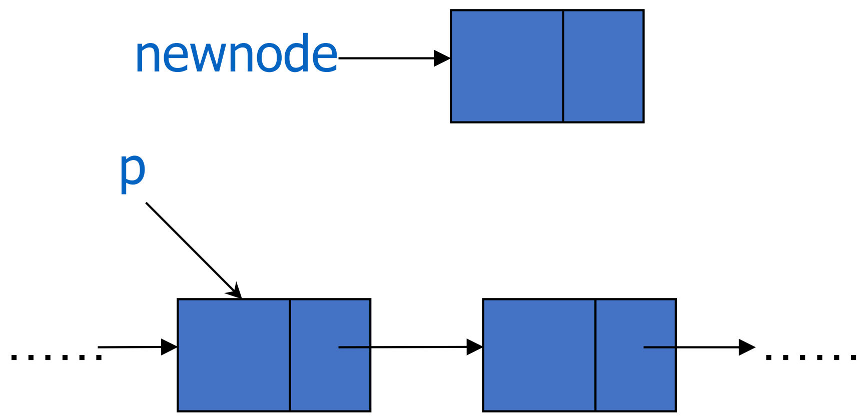
We don't care about the value of this node.

**Data nodes**

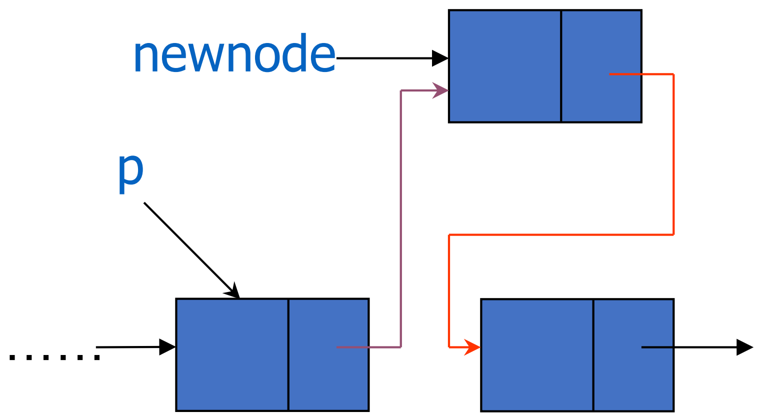
The value of these nodes are our data

# Dummy Header Node

- One case remaining for insert



before insert



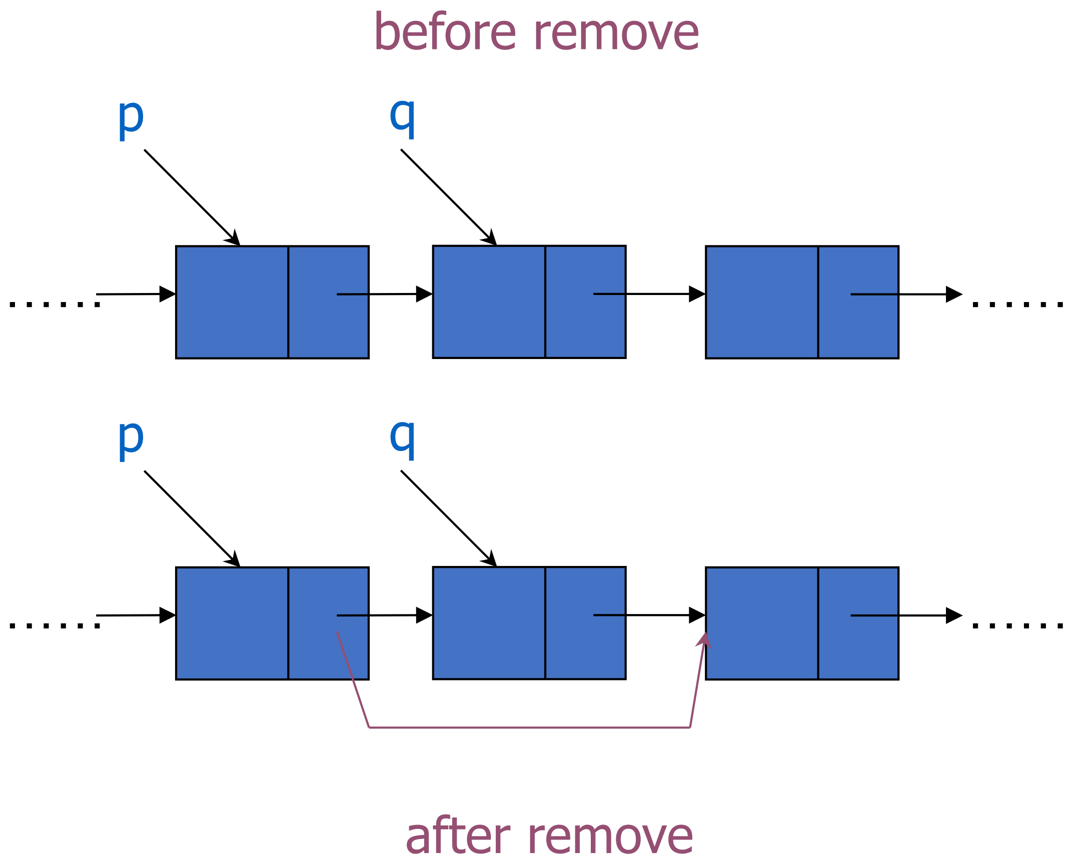
after insert

## 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;
        }
}
```

## Dummy Header Node

- One case remaining for remove

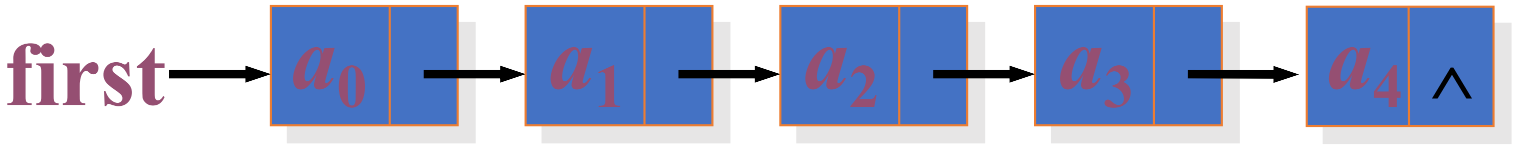


## Remove a node: With Dummy Header

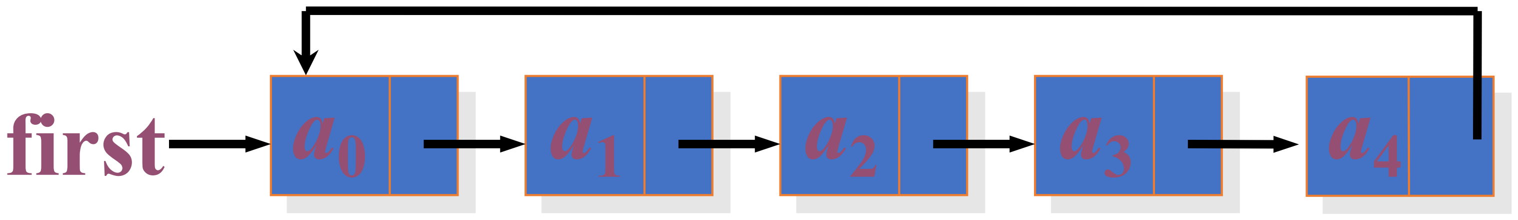
```
Void List::RemoveNode(ListNode* q)
{
        //remove at the beginning of a list
        if (q==first)
                first=first->next;
        else
        //remove not at the beginning
        {
                ListNode* p=first;
                while(p->next!=q)
                        p=p->next;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

## Circular Lists

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

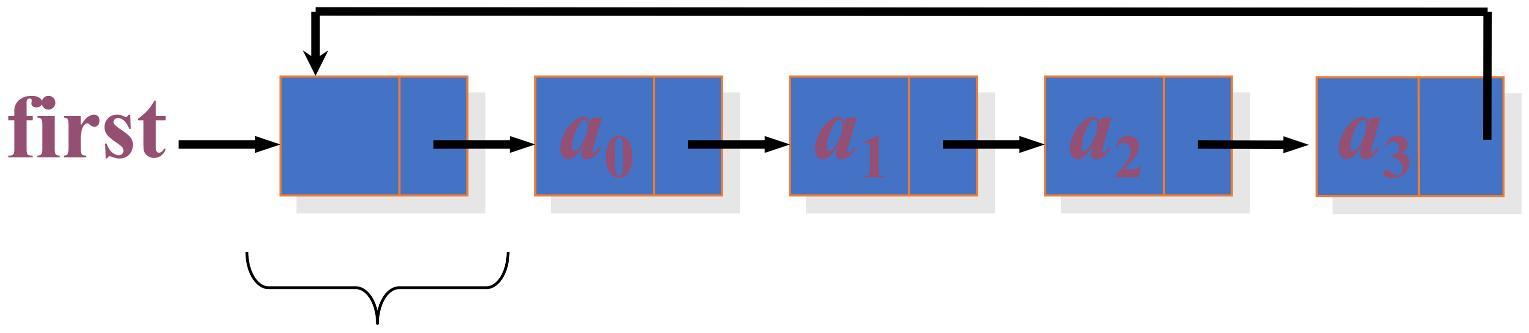


- If one extra link is allowed:



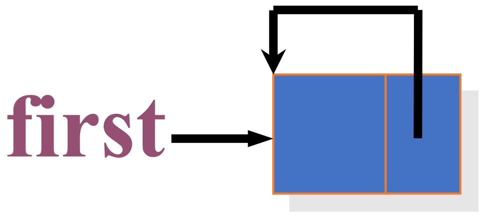
# Circular Lists

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



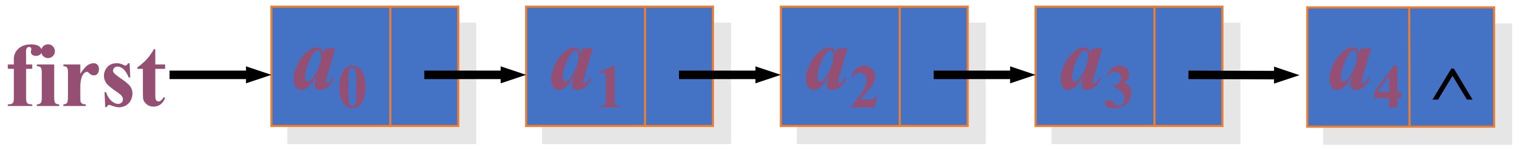
Dummy Header node

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

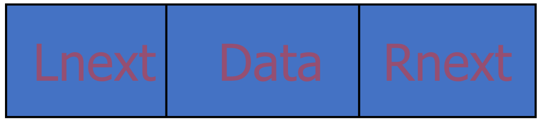


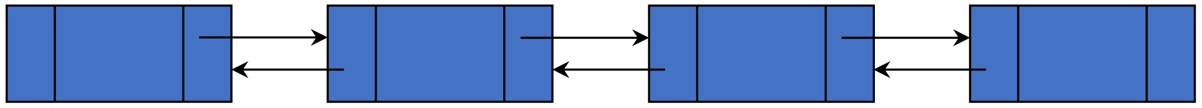
## Doubly Linked List

- 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 $a_4$



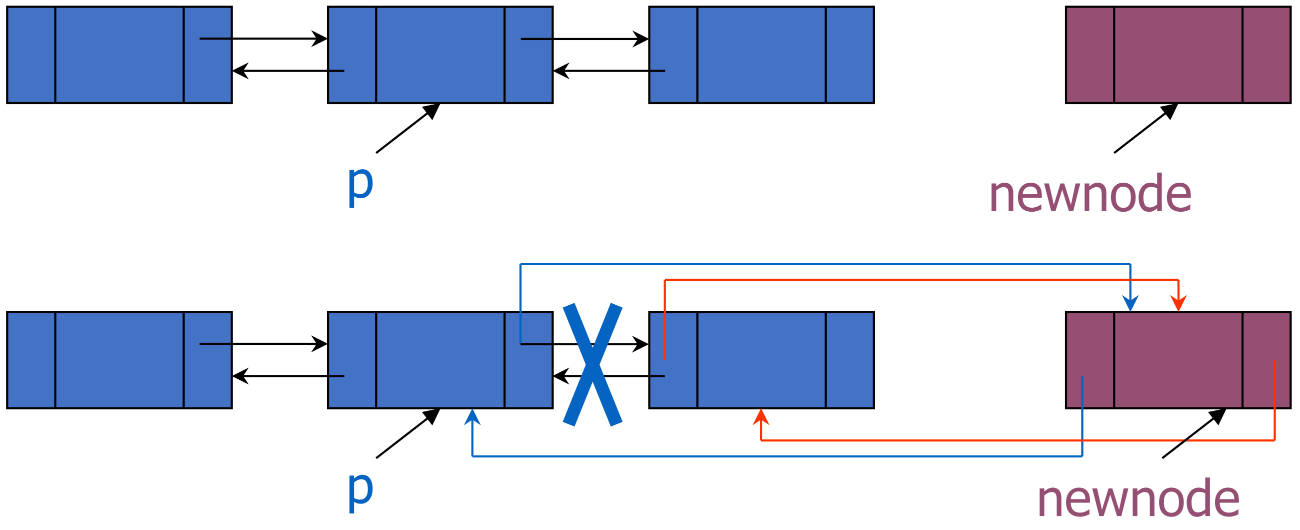
- If allowed to use more memory spaces, what to do?





# Doubly Linked List

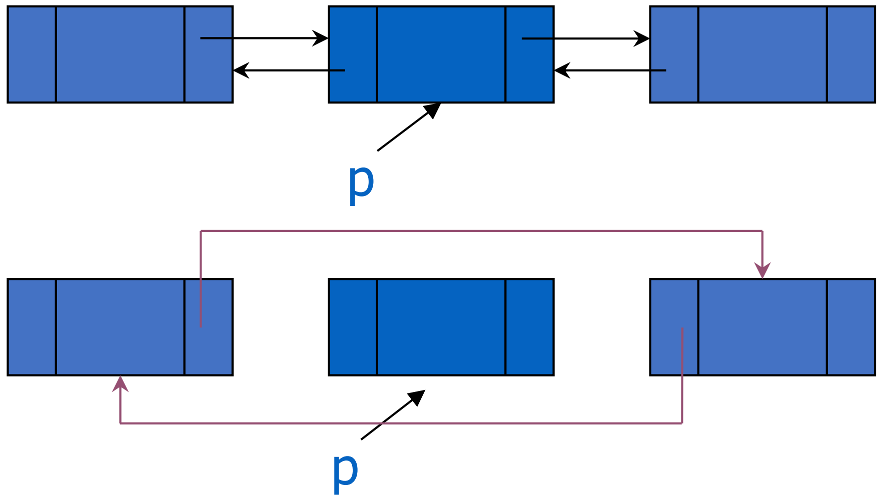
- To insert a node after an existing node pointed by p



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

# Doubly Linked List

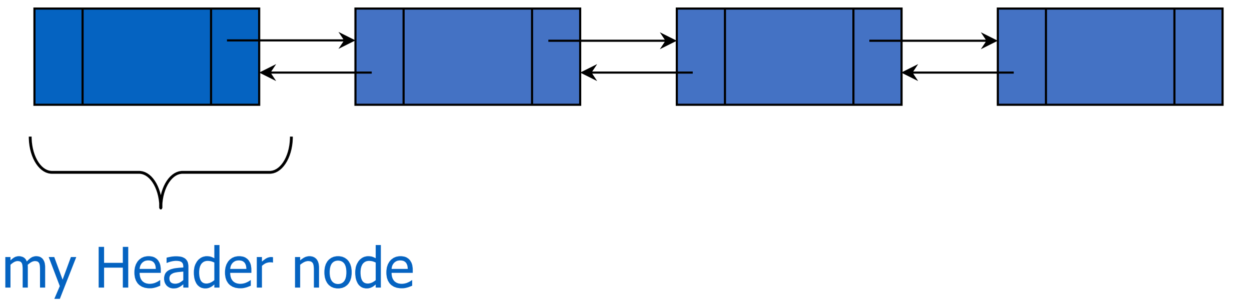
- To delete a node, we only need to know a pointer pointing to the node



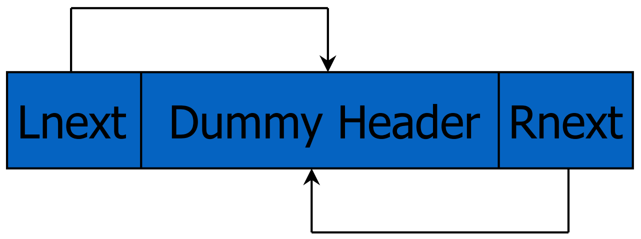
```
DoublyLinkedList::RemoveNode(ListNode *p)
{
        if(p->Lnext!=NULL) p->Lnext->Rnext=p->Rnext;
        if(p->Rnext!=NULL) p->Rnext->Lnext=p->Lnext;
}
```

## Further Extensions

- Doubly Linked List with Dummy Header



- Doubly Circular Linked List?



## 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:**

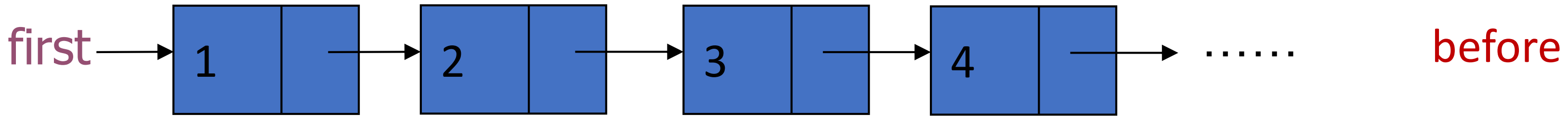
- Take up additional memory space for the links
- Accessing random parts of the list is slow. (need to walk sequentially)

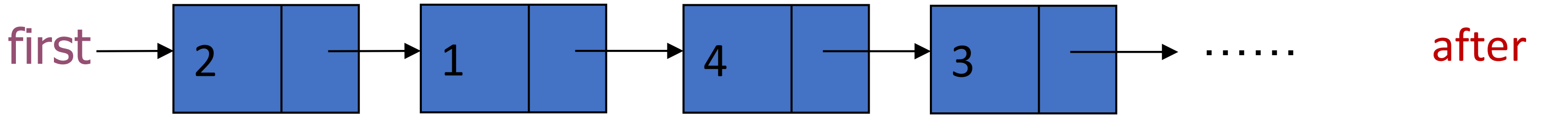
# Exercise 1

Swap two adjacent elements by adjusting only the links (and not the data) using singly linked lists

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

void swapAdjacentSingly(ListNode * first) {
            ...
}
```





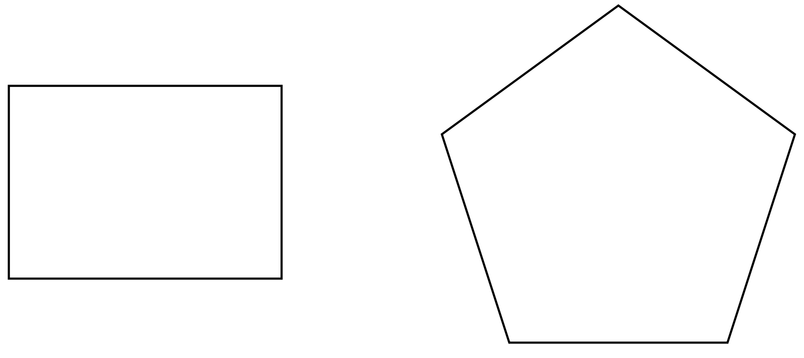
## Exercise 1

```
void swapAdjacentSingly(ListNode * first) {
    ListNode * current = first;

    while (current != nullptr && current->next != nullptr) {
        // swap
        ListNode* temp = current->next;
        current->next = temp->next;
        temp->next = current;

        // keep track of previous node
        if (current == first) {
            first = temp;
        } else {
            Node* prev = first;
            while (prev->next != current) {
                prev = prev->next;
            }
            prev->next = temp;
        }
        current = current->next;
    }
}
```

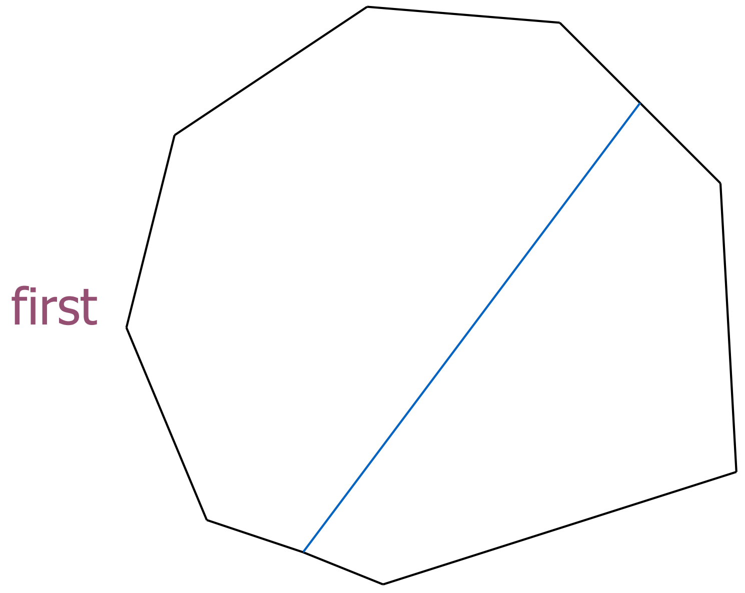
## Applications Representing Convex Polygons

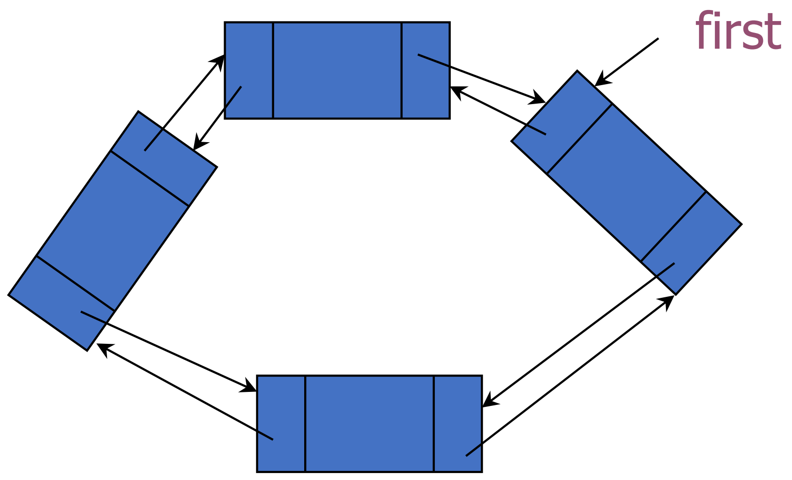


Polygon: A closed plane figure with *n* sides.

A convex polygon is a polygon in which

1) all interior angles are less than 180 degrees;
2) all line segments connecting any pair of points within the polygon lie entirely within the polygon

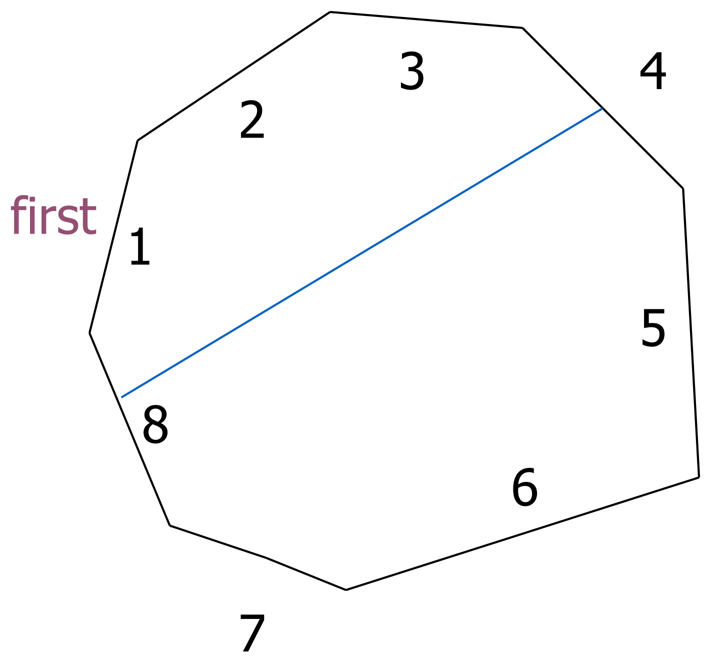


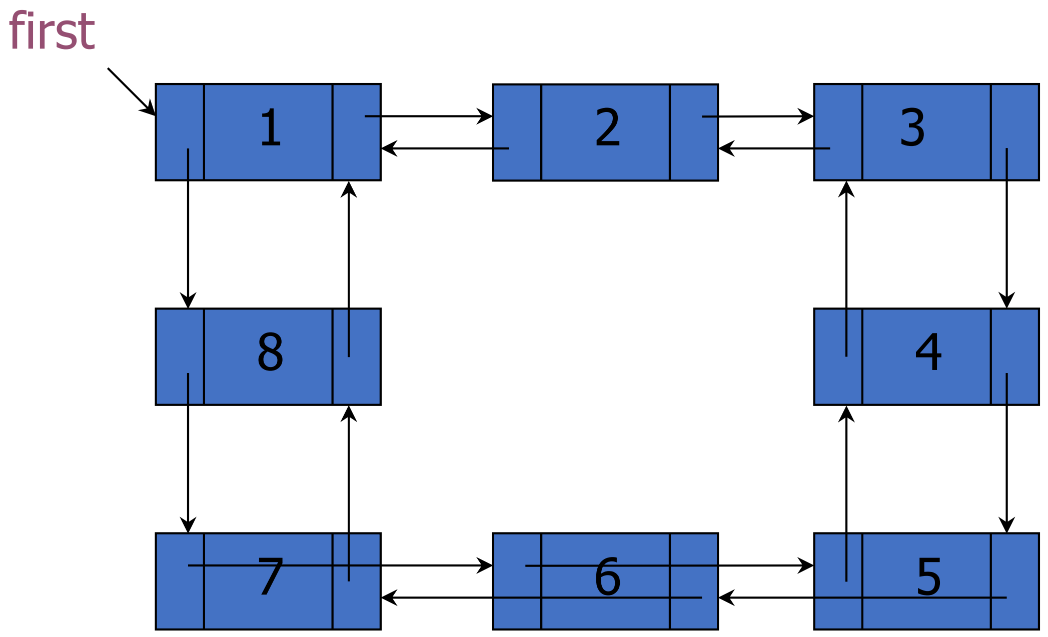


Every node represents a line

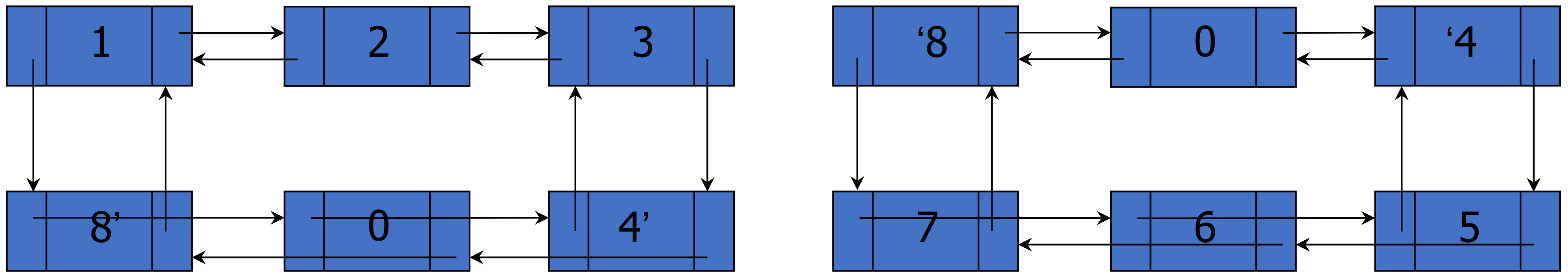
Easy to handle partition

# Applications Representing Convex Polygons





After Cut



## Learning Objectives

1. Explain the concepts of various linked lists and ADT
2. Able to implement linked list
3. Able to implement specific functions on linked list
4. Able to modify linked list structure for other applications

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