CS2351
Data Structures

Lecture 8:
Basic Data Structures I

About this lecture

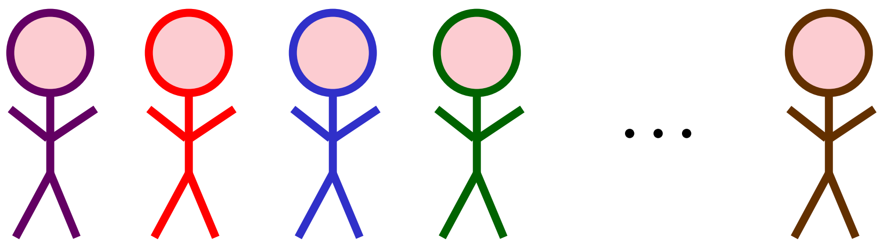
- Once we have learnt pointers, we can now define some basic, but very useful, data structures
- We will introduce three of them here:

1. List
2. Queue (also called FIFO queue)
3. Stack (also called LIFO queue)

List

List

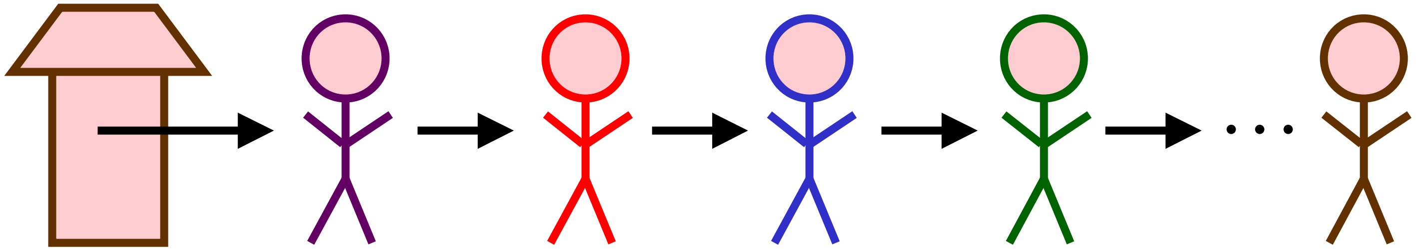
- A list (or linked list) is a data structure to represent a sequence of items, one after the other



A list of people

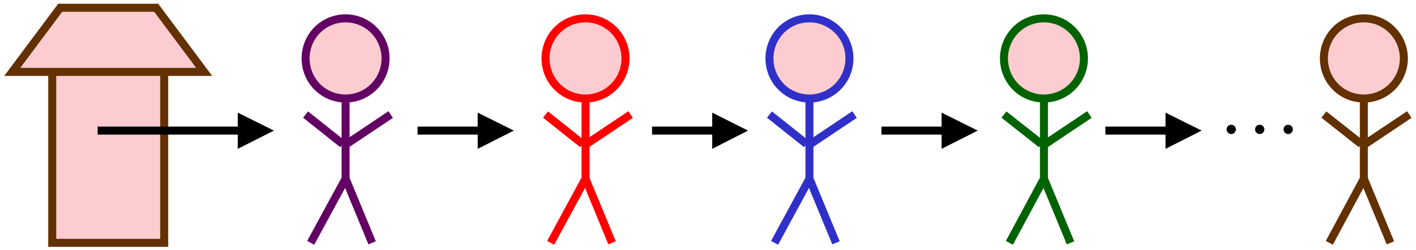
List

- Each item in the list points at the item immediately after it
- Usually, we keep an extra pointer, called head, to point at the first item



List

- Once the head of a list is known, we can traverse the list (from the beginning to the end) in linear time
- Usually, an item in the list is called a node



Implementing a List in C

- In C, we can first define a new type to represent a node :

```
struct node {
    ...
    ...
} ;
```

Implementing a List in C

- Since each node points to the next one, so we should have :

```
struct node {
   ...
   struct node *next ;
} ;
```

Implementing a List in C

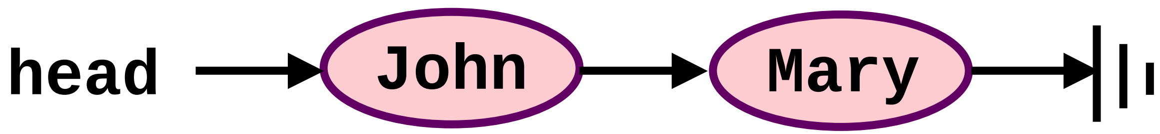
- Also, each node may contain some info
- Ex: To represent a list of people, a node may need to store the name of a person
- In this case, the definition may look like :

```
struct node {
  char name[80];
  struct node *next ;
} ;
```

Implementing a List in C

- Once the definition of a node is done, we can create a list

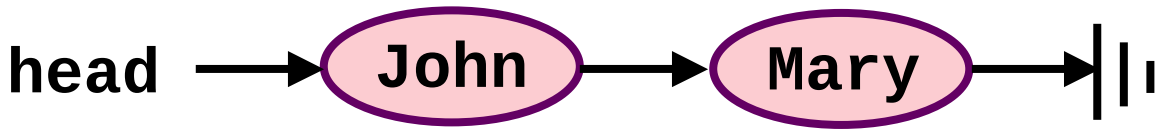
```
struct node x, y, *head ;

strcpy(x.name, "John");
strcpy(y.name, "Mary");
head = &x;  x.next = &y ;
y.next = 0;
```



Implementing a List in C

- Also, we can traverse a list easily

```
struct node *current ;
current = head ;
while ( current != 0 )
{
    printf("%s\n", (*current).name);
    current = (*current).next ;
}
```



Remark 1

- Recall that we have written something like

```
y.next = 0 ;
```

  to specify that y points to nothing

- In C, we often use NULL to replace 0, so as to show it indeed represents a location
- Then, we will write something like :

```
y.next = NULL ;
```

```
while ( current != NULL ) { ... }
```

Remark 2

- Recall that we have written something like

```
current = (*current).next ;
```

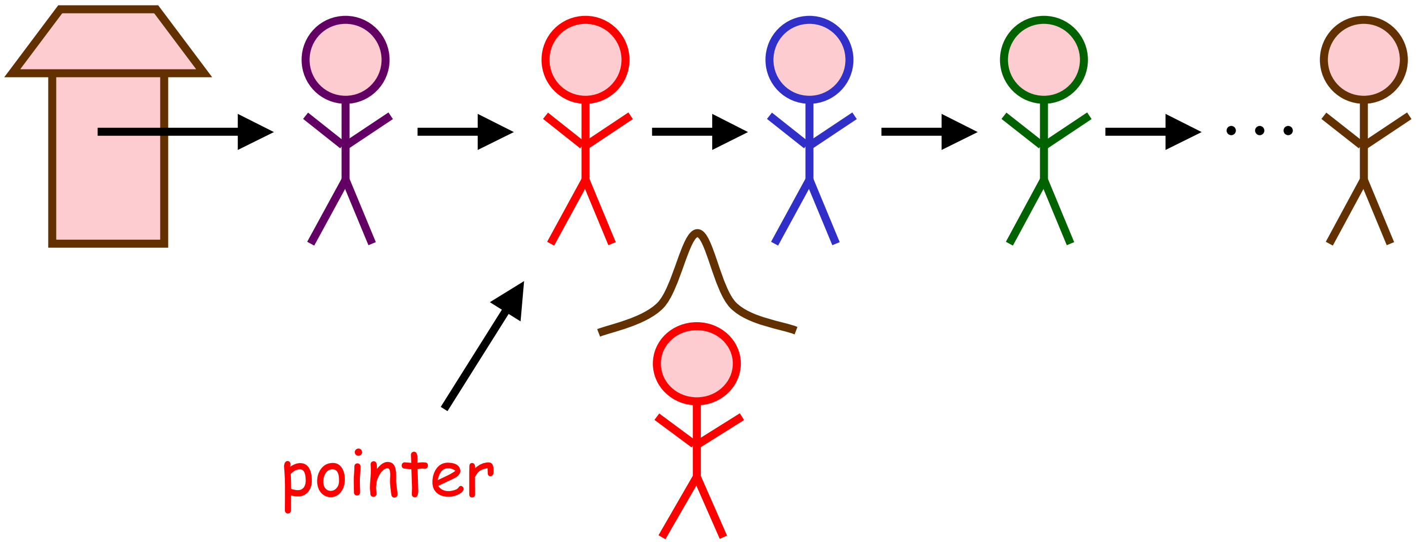
- The right hand side looks clumsy
- In C, we have a shorthand notation `->` (which looks like an arrow) to simply
- Instead of `(*current).next`, we write

```
current = current->next ;
```

- In general, `(*ptr).val` is exactly `ptr->val`

Insert in a List

- Suppose we have a pointer that points at a node X in the list
- Then, we can easily use this pointer to insert an extra node after X (How ?)



Insert in a List

- Let `current` be the pointer that specifies where to insert
- Let `y` be the extra node to be inserted
- Then, we can perform insert as follows:

```
y.next = current->next ;
current->next = &y ;
```

- Thus, if we know where to insert, only O(1) time is required !

Delete in a List

- Similarly, if there is a pointer that points at a node X, we can delete a node after X

```
if ( current->next != NULL )
{
    current->next
          = current->next->next ;
}
```

- Thus, if we know where to delete, only O(1) time is required !

Remarks for List Updates

Q: If we have a pointer that points at X, can we insert a node before X ?

A: Yes. We traverse from head, until we find a node Y that points to X in the list

- Y must be the node before X
- After that, we insert an extra node after Y

Q: Then, can we delete a node before X ?

A: Yes. (How ?)

Remarks for List Updates

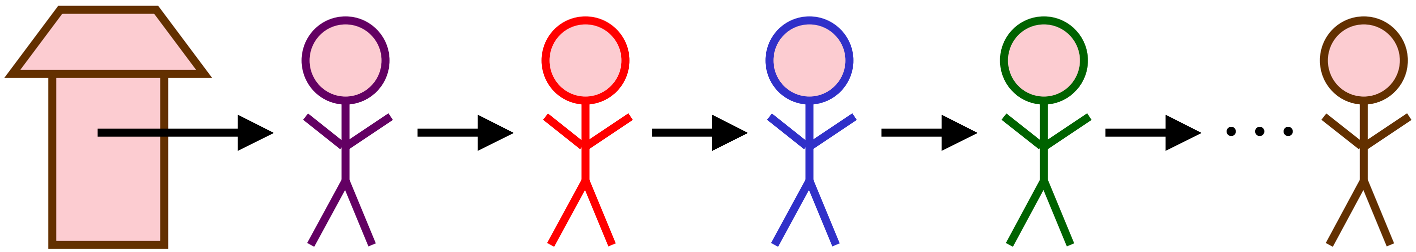
- Insert/delete before a node is tedious
  - In the worst case, it takes linear time !
- If we want to support such operations, we may use doubly linked list, so that each node has two pointers
  - one to previous node, one to next node

```
struct node {
   ...
   struct node *prev, *next ;
} ;
```

Queue

Queue

- A queue is a special kind of list where insertion is always at the end, and deletion is always at the front



Deletion always at the front

Insertion always at the end

Deletion in a Queue

- Since we have the head of a list, we can perform deletion easily (in O(1) time)

```
if ( head != NULL )
{
    head = head->next ;
}
```

- Here, we assume that in an empty queue, head is set to NULL

Insertion in a Queue

- To speed up the insertion, we will keep an extra pointer, called tail, that points at the last item in a queue
- Then, we can insert a node y in O(1) time without traversing the whole queue :

```
if ( head != NULL )
{
    tail->next = &y ;
    tail = &y;
}
```

Remarks for Queues

- Because we now maintain both head and tail pointers, we need to be careful in the boundary cases (when we insert a node in an empty queue, or delete the node to make the queue empty)
- The insert/delete operations in a queue are often called enqueue/dequeue
- Queue is also known as FIFO (first in first out) queue

Remarks for Queues

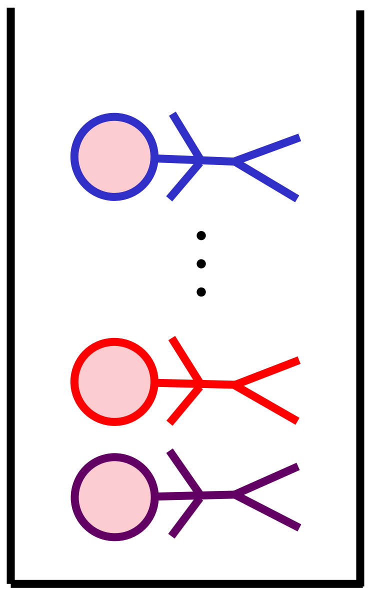
- To summarize the above, we may write a function for enqueue as follows:

```
void enqueue( struct node **head,
  struct node **tail, struct node *y )
{
  if ( (*head) != NULL ) // if not empty
  {  (*tail)->next = y ; (*tail) = y; }
  else
  {  (*head) = (*tail) = y ; }
}
```

Stack

Stack

- A stack is a special kind of list where insertion/deletion are always at the end
- Such an end is often called top



Insertion/Deletion always at the top

Deletion in a Stack

- We maintain a pointer, called top, to points at the top of the stack
- Since after deletion, we need to update top, each node should point at the previous node in the stack
- Then deletion is easily done (in O(1) time) :

```
if ( top != NULL )  // if not empty
{
    top = top->prev ;
}
```

Insertion in a Stack

- Insertion of a node y into the stack is also easy (done in O(1) time)

```
y.prev = top ;
top = &y ;
```

Remarks:

- Insertion/Deletion operations in a stack are often called Push/Pop
- Stack is also known as LIFO (Last in first out) queue

Practical Implementation

- In practice, we normally use an array to represent Queue or Stack

  Advantage: Each operation is faster

  (no need to keep next/prev pointers)

  Disadvantage: Wasted space / Overflow

- We will discuss further in the tutorial