CSD101: Introduction to computing and programming (ICP)

Linked structures I

- Structure instances can be linked in a chain (unidirectional or bidirectional) by using pointers to structures via a self referential mechanism.

```
struct Node {
  int data;
  struct Node *next;
};
// next points to the next structure instance or is NULL
```

- While declaring a `Node` we have to use the *tag* style of declaration and cannot use `typedef`.
- The above is called a **singly linked list** - unidirectional link.
- We can define a bidirectional or **doubly linked list** that has pointers to the next node and also the previous node.

Linked structures II

```
struct Node {
  int data;
  struct Node *next, *prev;
};
// next points to the next Node instance or is NULL
//prev points to the previous Node instance or is NULL
```

Data structure or data type

Definition 7

A data structure (or data type) is a compound value that contains other values and that has certain well defined behaviour.
The behaviour is concretely defined in terms of various operations on the data structure (typically creation, addition, deletion, modification, emptyness, etc.) and properties that these operations must follow.
Data structures are used to store data and implement algorithms efficiently.

Note that the word *structure* in data structure is not related to **C** structures. We use **C** structures and linking to implement some data structures. For example the *singly/doubly linked lists* are data structures.

Stack - data structure I

- A **stack** is a data structure with Last in first out (or LIFO) behaviour. That means the element added last to the stack comes out first.
- A stack has 3 operations defined on it apart from create.
  `Stack *push(Stack *st, int info` - adds `info` to the stack and returns a pointer to the new stack.
  `Stack *pop(Stack *st, <element-type> *info)` - removes the top element in the stack and returns a pointer to the popped stack. It also returns the popped element via a pointer to the element in the argument list.
  `int isEmpty(Stack *st)` - returns `true` if stack st is empty else it is `false`.
  `Stack *create()` - creates an empty stack and returns a pointer to it.

Stack - data structure II

- Stacks are used in many places. One important use is in managing function calls, returns and arguments to functions.
- We will implement a stack using a list.

Queue - data structure I

- A **queue** is familiar to us from daily life. It has first in first out behaviour (or FIFO). That is any data item that enters the queue first exits first.
- A queue also has 3 operations apart from create.
  `Queue *addq(Queue *qp, int info` - adds `info` to the queue and returns a pointer to the new queue.
  `Queue *deleteq(Queue *qp, <element-type> *info)` - deletes the item at the front of the queue and returns the resulting queue. The info in the deleted item is returned in `info`.
  `int isEmpty(Queue *qp)` - returns true if queue qp is empty and false otherwise.
  `Queue *createq(void)` - creates and returns an empty queue.

Queue - data structure II

- We will implement a queue using a singly linked list. But first we add an operation:

  `struct Node *deleteAtEnd(struct Node *listp, int *info)`

  to a `list`. However, this implementation is inefficient since it has to traverse the entire list to reach the end of the queue. To make it efficient an option is to create a Queue data structure with an embedded list with pointers to the beginning and end of the list. Add to the end of list and delete from the beginning of the list