CSC 301: Data Structures and Algorithms

Fundamental Data Structures

Denis L. Nkweteyim

Outline

- Pointers

- Structures

- Arrays and dynamic memory allocation

- Linked lists

- Stacks and queues

- Strings

Pointers (brief revision)

- Pointers

  - Useful for accessing memory and manipulating memory addresses

  - Declaration

    - Similar to variables, except that an asterisk (*) is placed before the identifier

    - Example

      - `int i, *p;` //variable i is of type int
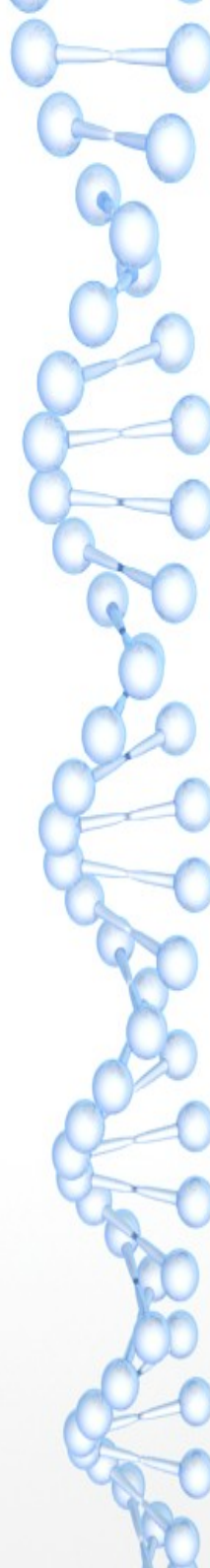   //variable p is of type pointer to int

Pointers (brief revision)

- Data (of the same type as the pointer) can be assigned to pointer variables

- Example

  - p = &i;   //OK, if i is a variable of a certain type, and p is a pointer to data of that type

- The value 0 or equivalently NULL can also be assigned to a pointer of any type

  - In such a case, the pointer points to nothing, not even zero

  - Examples

    - p = NULL; and p = 0; are equivalent, if p is a pointer variable

Pointers (brief revision)

- Dereference operator, *
    - Used to access the value that a pointer variable points to

```
int a, b;
int *p;
a = 5; b = 4; p = &a; //p points to address of a
printf("*p = %d\n", p); /* p = 5 */
printf("a = %d\n", a); /* a = 5 */
*p = 20;
printf("*p = %d\n", p); /* p = 20 */
printf("a = %d\n", a); /* a = 20 */
```

Two uses of pointers

- Manipulate contents of memory (variable) from within a called function

- In functions that need to compute and return more than one value

Pointers and Function example

```c
#include <stdio.h>
void swap(int *, int *);
int main(void) {
   int x = 10, y = 20;
   printf("Before calling function\n");
   printf(" x = %d\n", x); printf(" y = %d\n\n", y);
   swap(&x, &y);
   printf("After calling function\n");
   printf(" x = %d\n", x); printf(" y = %d\n", y);
}
void swap(int *p, int *q) {
   int tmp;   tmp = *p; *p = *q; *q = tmp;
}
```
Program Output
```
Before calling function
 x = 10
 y = 20
After calling function
 x = 20
 y = 10
```

Structures

- Structures (structs in C)

  - Allow us to define collections of data that can be manipulated as a single unit

  - We can

    - Refer to components of a structure

    - Assign one struct to another

      - All the components of one struct copied to corresponding components of the other

Defining a Structure

- Simple example
  - Student ID, student name, and score for a test
    - **struct st_rec** is a user-defined data type
    - Like with all data types, variables of the type can be declared

```
struct stu_rec {
    int id;
    char name[15];
    int score;
};
struct stu_rec pers1, pers2;
```

Alternative approach to define a struct

- Using typedef
  - A struct can be defined without a tag name
  - At the time of definition, variables can be declared
    - In this example, the struct is called stu_rec, not struct stu_rec as in the previous example

```
typedef struct {
    int id;
    char name[15];
    int score;
} stu_rec;
```

Referring to components of a struct

- Structure member operator (.)
    - Used to access struct members via simple variables
    - Example: `pers1.id = 2;`

- Structure pointer operator (->)
    - Used to access struct members via pointer variables
    - Example: `pers2_ptr -> id = 10;`

Example struct program

```c
#include <stdio.h>
#include <string.h>

int main(void) {
    struct stu_rec {
        int id;
        char name[15];
        int score;
    };
    struct stu_rec p1, p2;
    struct stu_rec * p2_ptr;

    p2_ptr = &p2;

    p1.id = 2;
    strcpy(p1.name, "Peter Jones");
    p1.score = 85;
    printf("\nName: %s\nID: %d\nScore: %d\n", p1.name, p1.id,
            p1.score);
    p2_ptr->id = 10;
    strcpy(p2_ptr -> name, "John Brown");
    p2_ptr -> score = 75;
    printf("\nName: %s\nID: %d\nScore: %d\n", p2_ptr->name,
            p2_ptr->id, p2_ptr->score);
}
```

Modify the program

1. Modify the field for score to handle not just one score but 5. Choose the best data structure for this purpose

2. Modify your program in (1) above further as follows

   a) Use one array, and not separate variables, to handle data for N students, where N is a constant you define

   b) Compute the average score earned by each student

   c) Compute the average score earned by the N students for each subject, and for all the courses put together

struct Example 2

- Program on next slide
  - Uses struct to define two points in 2-D space
  - Find the distance between the points
- Modifications to make
  - Generate N random points with x,y coordinates less <= 1 in 2-D space (N is a constant)
    - Note: 1.0 * rand()/RAND_MAX; generates a random num between 0 and 1. You need to include stdlib.h
    - Compute distance between each pair of points
  - Use the function below to convert x,y coordinates to polar coordinates
    - Use some of the points created above to test the function

(x,y)

θ
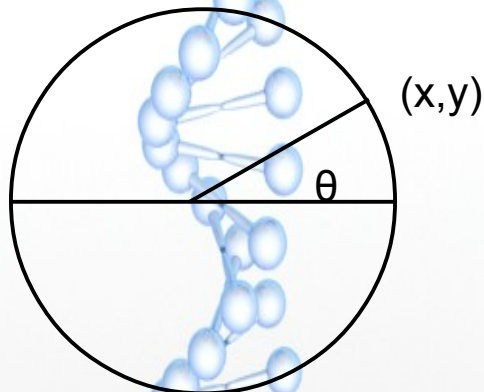
```
polar (float x, float y, float *r, float *thetha){
*r = sqrt(x*x + y*y);
*thetha = atan2(y, x);
}
```

struct Example 2

```c
#include <stdio.h>
#include <math.h>

typedef struct {
  float x;
  float y;
} point;
float distance(point, point);
int main(void) {
    point a, b;
    a.x = 5; a.y = 10;
    b.x = -5; b.y = 5;
    printf("Distance between (%.2f,%.2f) and (%.2f,%.2f) is
            %.2f\n",a.x,a.y,b.x,b.y,distance(a,b));
}
float distance(point a, point b) {
  float dx = a.x - b.x, dy = a.y - b.y;
  return sqrt(dx * dx + dy * dy);
}
```
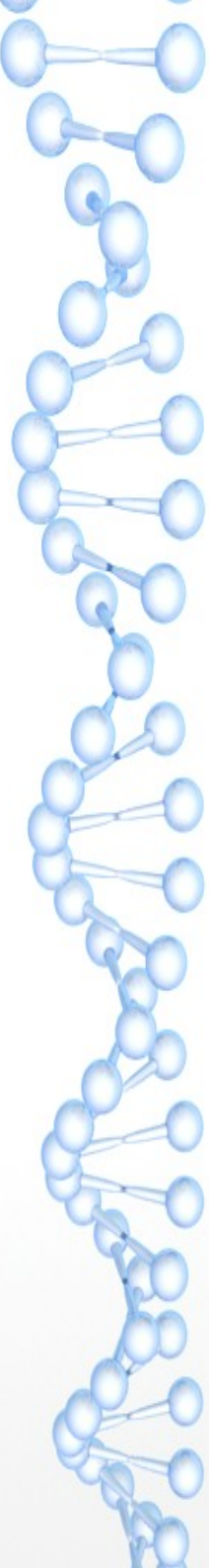
Arrays and Pointers

- Array

  - Contains homogeneous data

  - 1-D array

    - Contains list of items

  - 2-D array

    - Tabular structure comprising rows and columns

  - 3-D array

    - Cube-like structure

    - Comprises several 2-D arrays arranged along a third dimension

  - Etc.

Array example – matrix addition

- Use 2-D arrays to represent matrices

- Matrix addition

    - Matrices **a** and **b** of dimension N x M to get Matrix **c**

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    c[i][j] = a[i][j] + b[i][j];
```

Array example – matrix multiplication

- Code below multiplies 2 NxN matrices

  - Modify to multiply an mxn matrix by an nxp matrix

```
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      for (k = 0, c[i][j] = 0.0; k < N; k++)
         c[i][j] += a[i][k] * b[k][j];
```

Arrays and Pointers

- In C

  - An array of length N is indexed from zero to N-1

  - Array name is equivalent to a pointer to the first element of the array

    - Hence, can use C's pointer arithmetic to arrays

    - Example

      - If *p points to the first object, *(p+1) points to the second object, *(p+2) to the third object, etc
      - Applying these ideas to array a, *a refers to a[0], *(a+1) to a[1], *(a+i) to a[i]

  - Pointers to arrays can be passed to functions that process arrays

    - A lot more efficient than having to pass the complete array to functions (especially if the array is large)

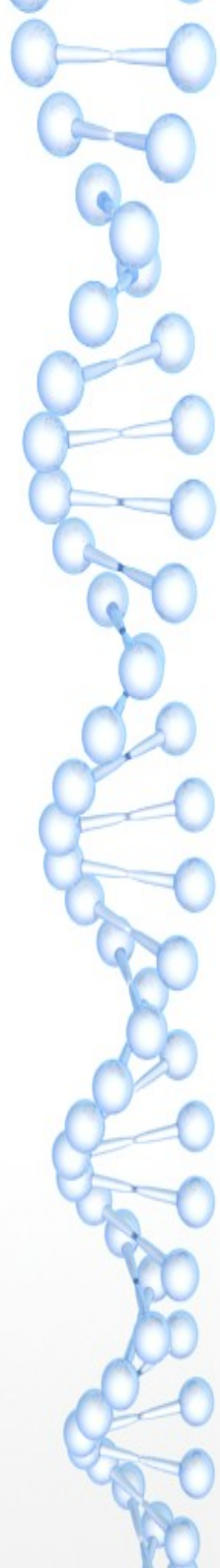# Arrays and Pointers

- Declaring Arrays
  - Size is required at the time declaration
  - This is static declaration
    - Memory is reserved at the time the array is declared
    - Memory cannot be used for any other purpose for as long as the program is running
  - But the array size is usually not known at the time the program is written, and only becomes known at the time the program is run
  - Dynamic memory allocation overcomes this problem by providing the mechanism to allocate the required memory at run time
  - When dynamically allocated memory is no longer needed, it can be freed to be re-used elsewhere in the program

Arrays and Pointers

- Two functions in the standard library for dynamically allocating memory for dynamic memory allocation

  - calloc() - contiguous allocation

  - malloc() - memory allocation

- Prototypes

  - void *calloc(size_t, size_t);

  - void *malloc(size_t);

    - Data type size_t is defined in stdlib.h, and is typically an unsigned integer

    -

    - For calloc(): The product of the two parameters shoud be equal to the amount (in bytes) of memory to be allocated

    - For (malloc): The parameter should be equal to the amount (in bytes) of memory to be allocated

Dynamic Memory Allocation Example 1

- Example simulates a coin tossing experiment

  - Experiment is run N times

  - For each of the N experiments, coin is tossed M times

  - Output is plot of number of heads for each of the N experiments

    - To economize screen space, one asterisk corresponds to 10 heads

Dynamic Memory Allocation Example 1

```c
#include <stdlib.h>
#include <stdio.h>
/*funtion heads returns 1 if head, 0 if tail*/
int heads() {
  return rand() < RAND_MAX/2; /*head if rand num <RAND_MAX/2, else tail*/
}
int main (int argc, char *argv[]) {
  int i, j, cnt;
  int N = atoi(argv[1]);
  int M = atoi(argv[2]);
  /*f is an array whose memory is dynamically allocated*/
  int *f = malloc((N+1)*sizeof(int)); /*up to N+1 heads, (0 to N)*/
  for (j = 0; j <= N; j++)
    f[j] = 0; /*initialise frequency of heads*/
  for (i = 0; i < M; i++, f[cnt]++) /*experiment repeated M times*/
    for (cnt=0, j=0; j <= N; j++) /*N+1 possible values for each experiment*/
      if (heads())
        cnt++;
  for (j = 0; j <= N; j++) {
    printf("%2d ", j);
    for (i = 0; i < f[j]; i+=10)
      printf("*");
    printf("\n");
  }
}
```

Dynamic Memory Allocation
Example 1

```
dyn 20 3000
  0
  1
  2 *
  3
  4 *
  5 **
  6 ********
  7 ****************
  8 ****************************
  9 ********************************************
 10 **************************************************************
 11 **********************************************************
 12 ********************************************
 13 ******************************
 14 ***************
 15 **********
 16 **
 17 *
 18 *
 19
 20
```

Array of Arrays

- Multidimensional array
  - Can be implemented as arrays of arrays
  - Provides added flexibility to programs
    - e.g., arrays of arrays that differ in size
  - Treat as 1-D array (static or dynamic) of pointers, and use dynamic memory allocation to allocate memory for the contents of each cell

Array of Arrays Example

- Notes
    - t is a pointer to *pointer to int*
        - Recall array name is pointer to base of array
            - Hence, t is an array whose cells are *nameless* arrays (pointers to int)
                - More correctly, the name of the array the first cell of t points to is t[0], second cell t[1], etc.
    - In the example, each row points to a 1-D array of c ints
        - We could choose to vary the size of each of these int arrays

```
int **malloc2d(int r, int c) {
    int i;
    int **t = malloc(r * sizeof(int *)); /*r rows*/
    for (i = 0; i < r; i++)
       t[i] = malloc(c * sizeof(int));
    return t;
}
```

Other problems with static arrays

- Arrays are a great tool
  - Very easy to use
- But apart from need to know array size in advance, there are certain normally trivial operations that are hard to do with arrays
  - Inserting items in the middle of an array
    - Requires right-shifting items to the right of the insertion point
  - Deleting items from an array
    - Requires left-shifting items from the right of the insertion point

Student Homework

- Do the following assignment to appreciate the difficulties in processing static arrays
  - Make your program as rigorous as possible. Plan to spend about 90 minutes in the exercise

Write a program that does the following:
- create an integer array a of size 100 and initializes each cell of the array to -1
- store 20 randomly generated integers in the first 20 cells of the array
- call a function called display to display the contents of the array up to the first cell that contains -1 (do not forget to implement function display)
- write a function add that generates 10 random integers and stores them starting at index 5 of array a. Note that you need to shift down the 15 numbers starting at index position 5 ten places before inserting the generated numbers. Call function add from main, then call function display from main, to display the contents of the updated array
- write a function that deletes the 5 numbers at array index 15 to index 19. Note that you need to shift up all valid numbers starting at index 20, then fill the gaps created with -1.

When I ran my program, I got the following output:
Initial Array
383 886 777 915 793 335 386 492 649 421 362 27 690 59 763 926 540 426 172 736

10 newly generated numbers
211 368 567 429 782 530 862 123 67 135

Array after inserting generated numbers
383 886 777 915 793 211 368 567 429 782 530 862 123 67 135 335 386 492 649 421 362 27 690 59 763 926 540 426 172 736

5 Deleted Items
335 386 492 649 421

Array after deleting numbers
383 886 777 915 793 211 368 567 429 782 530 862 123 67 135 362 27 690 59 763 362 27 690 59 763 926 540 426

Linked Lists

- Linked lists overcome the problems with arrays

  - Need to know array size (for static arrays)

    - Not a problem for dynamically created arrays

  - Problem of deletion of items in array

  - Problem of insertion of items in array

- There is a trade-off

  - We shall see that linked lists do not have the flexibility of easy of access to array elements by simply stating the array index

    - With linked lists, you can only access list elements sequentially, one after the other until you get to the item you want

Linked List Data Structure

- Linked list

  - A set of items where each item is part of a *node* that also contains a **link** to a *node*

  - Notes

    - Nodes defined in terms of references to nodes
      - Hence, linked lists are also referred to as *self-referential* structures
    - A node's link may refer to a different node, or to the node itself
      - Hence, linked lists can sometimes be *cyclic* structures

  - Basic elements

    - Pointers used for links

    - Structures used for nodes

  - Various configurations of nodes possible

Linked List Data Structure

- Nodes
    - Contain the data we are holding in the list

A Linked list made up of a number of "date" records/structures

- Nodes
    - Most times, for sake of simplicity, we shall only use simple data on node
        - See example below right
    - Important to remember that in real life, nodes are much more complex data structures than simple data types
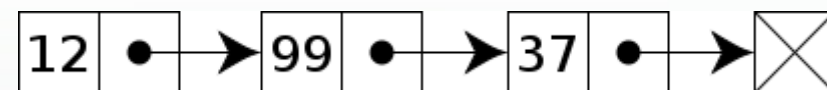
Examples of configuration of linked list nodes

1) Singly linked list

- Possible to move forward through the list, but not backward

```
12 • → 99 • → 37 • → ⊠
```
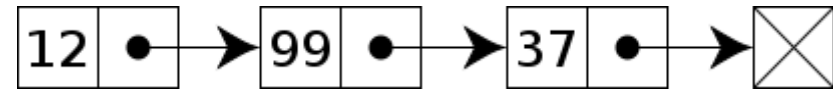
2) Doubly linked list

- Allows for movement both forward and backward through the list

```
⊠ ← • 12 • ⇄ • 99 • ⇄ • 37 • → ⊠
```
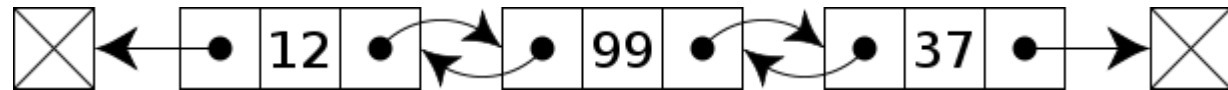
3) Circular linked list

- Nodes arranged in a cycle
- Last node is linked back to the first

```
12 • → 99 • → 37 •
```
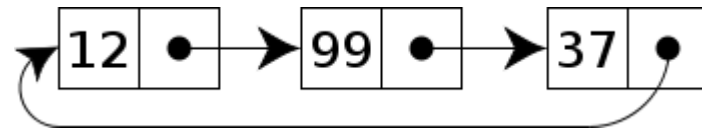
Creating a linked list – hard way

- Consider code for `struct` definition on the right

  - Has all the ingredients for the definition of a singly linked list

    - Field `next` is a pointer to `struct list`

      - Enables the link from one element record to the next

    - Reminder: data field could be any data structure, complex or simple

- Variables can now be declared and values assigned to them (top picture)

- Links can also be created (bottom picture)

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

```
struct list a, b, c;
a.data = 1; b.data = 2; c.data = 3;
a.next = b.next = c.next = NULL;
```

a
| 1 | NULL |

b
| 2 | NULL |

c
| 3 | NULL |

```
a.next = &b; b.next = &c;
```

a
| 1 | NULL |

b
| 2 | NULL |

c
| 3 | NULL |
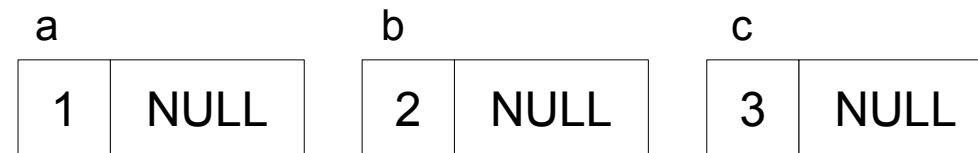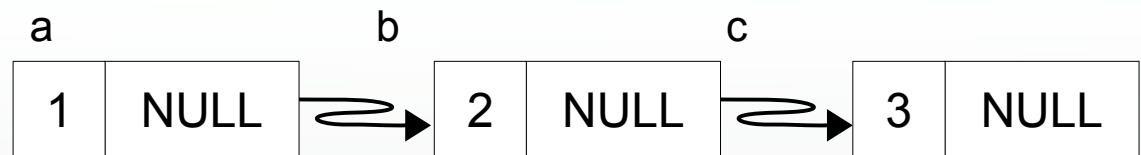
Creating a linked list – hard way

- Retrieving data from the list (examples)

```
a.next -> data same as b.data (has value 2)
a.next -> next -> data same as c.data (has value 3)
```

- Problem with linked list definition above

  - One variable for each node

    - Too tedious especially if there are several nodes in the list

  - More common (and convenient) to dynamically allocate the memory that a linked list needs

Creating a linked list – the usual way

- Notes

  - Possible to declare a pointer variable to a `struct` (`node`) even before the structure is defined

  - To add a new node, memory has to be allocated for it

    - `Link x = malloc(sizeof *x);`

```
typedef struct node *Link;
  struct node {
  int item;
  struct node *next; //alternatively, Link next;
};
```

Creating a linked list – the usual way

- The last (or tail) node in a singly linked list
  - Usually handled in a special way so that when the list is traversed, the program may know when the end of the list is encountered
  - Examples: Use
    - A null link that points to no node
    - A dummy node that contains no data
    - Points back to the first node, creating a circular list
- First (or head) node is also special
  - May contain valid data
  - May be used as a pointer to the first valid data node
  - Whichever convention is used depends on programmer preference, efficiency of the algorithm, etc.

Singly linked list example

```c
#include <stdio.h>
#include <stdlib.h>
  struct st_info {
  int id; float score1, score2;
};
typedef struct st_info ST_INFO;
typedef struct node *LINK;
struct node {
  ST_INFO stud; LINK next;
};
void list_add(LINK *head,ST_INFO st_info);
ST_INFO get_info(ST_INFO stud);
void list_prn(LINK head);
int main(void) {
  int i; LINK h;  ST_INFO st;
  h = NULL; /*empty list*/
  printf("Enter Student ID (int), Mark 1 (float), and Mark 2 (float) for four
          students");
  for (i = 0; i < 4; i++) {
    st = get_info(st); list_add(&h, st);
  }
  list_prn(h);
}
```

Singly linked list example

```c
ST_INFO get_info(ST_INFO stud) {
  printf("\nStudent ID: ");scanf("%d",&stud.id);
  printf("Score I: ");scanf("%f",&stud.score1);
  printf("Score II: ");scanf("%f",&stud.score2);
  return stud;
}
void list_add(LINK *head,ST_INFO st_info){
  LINK tmp;
  /*allocate space for new node*/
  if ((tmp = malloc(sizeof (*tmp))) == NULL) {/*out of memory - exit*/
    printf("\nNot enough memory");
    exit(1);
  }
  tmp -> stud = st_info;
  tmp -> next = *head;
  *head = tmp;
}
void list_prn(LINK head) {
  LINK tmp = head;
  printf("\n%12s%9s%10s","Student ID","Score I", "Score II");
  while (tmp != NULL) {
    printf("\n%12d%9.2f%10.2f",tmp->stud.id,tmp->stud.score1,tmp->stud.score2);
    tmp = tmp->next;
  }
  printf("\n");
}
```

Singly linked list example – sample output

```
Enter Student ID (int), Mark 1 (float), and Mark 2 (float) for four students
Student ID: 4
Score I: 67
Score II: 80.5

Student ID: 4
Score I: 45
Score II: 66

Student ID: 7
Score I: 39
Score II: 56

Student ID: 6
Score I: 57
Score II: 55

  Student ID   Score I   Score II
          6     57.00      55.00
          7     39.00      56.00
          4     45.00      66.00
          4     67.00      80.50
```

Element deletion from a singly linked list

- To remove a node following node **x**

  - Set **t** to point to the node to be removed

  - Change x's link to point to t->next

  - Note

    - Although we effectively bypass the deleted node, the memory allocated to it is not freed

    - To make that memory available for reuse, the free() function can be invoked

- The following two statements will delete (bypass) the node following node x:

  - t = x->next;

  - x -> next = t -> next;

- or equivalently,

  - x -> next = x -> next -> next;

- The first version is better if there is need to free the bypassed node using the statement: `free(t);`
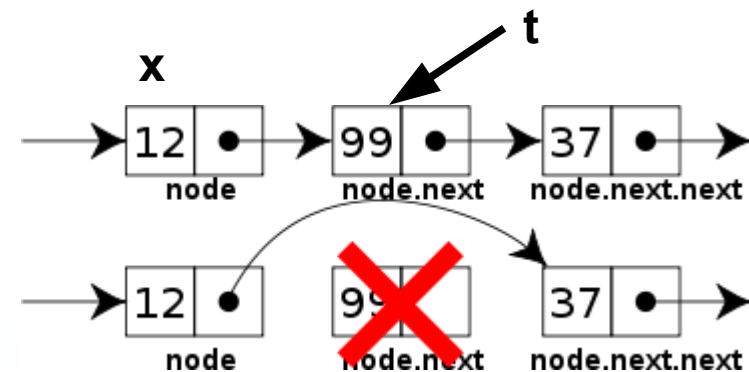
Element insertion in a singly linked list

- To insert a node **t** into a linked list at a position following another node **x**,

  - Set **t -> next** to **x -> next**

  - Set **x -> next** to **t**

- The following statements are used:

  - t -> next = x -> next;

  - x -> next = t ;
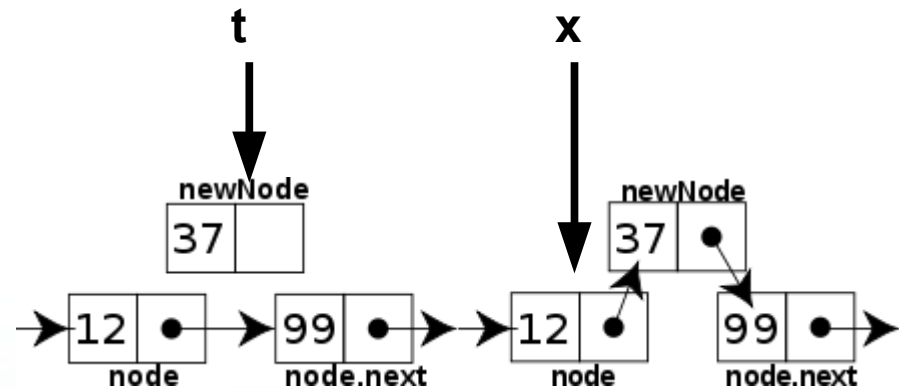
Doubly linked list

· Allow for movement both forward and backward through the list

- Each node has

  · A next link

    - Chains to next node on the right
    - Enables traversal to the right

  · A previous link

    - Chains to next node on the left
    - Enables traversal to the left
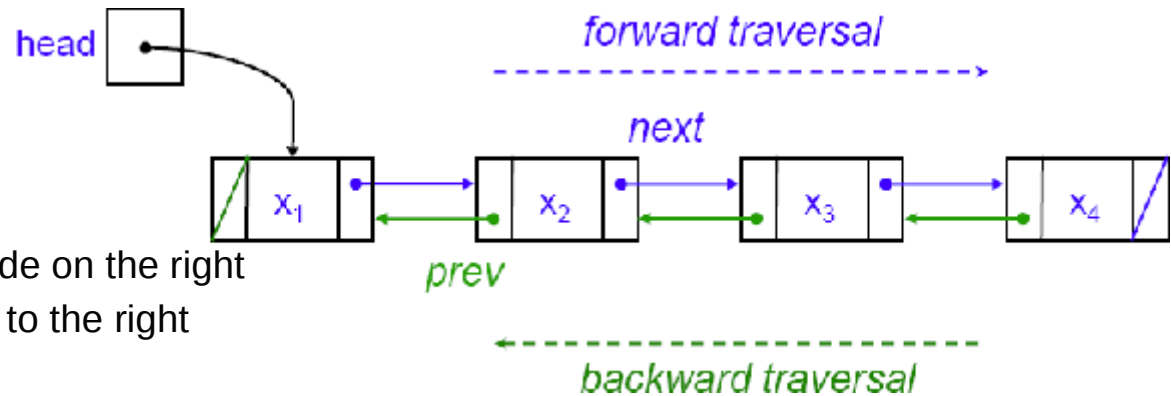
- Note that for a doubly linked list, pointers x, x->next->prev, and x->prev->next all point to the same node
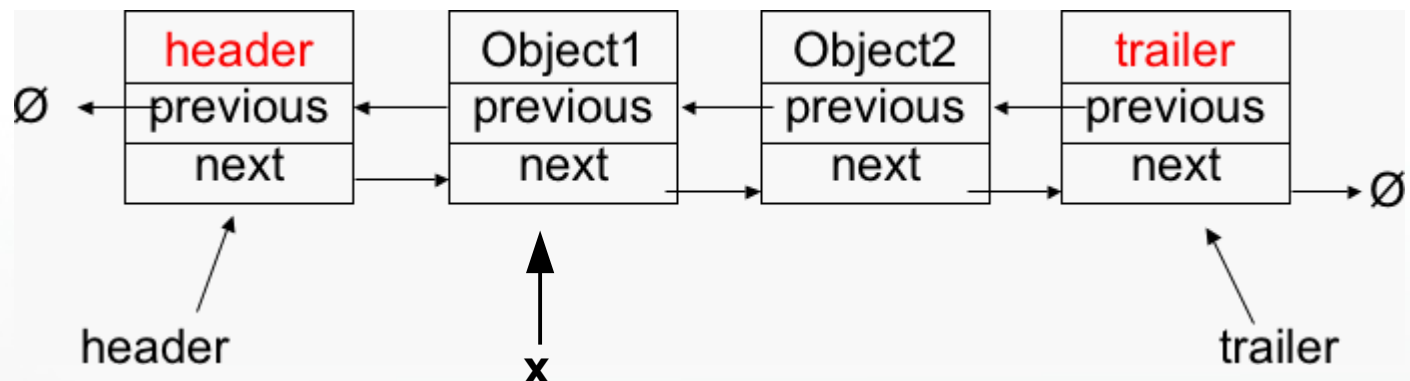
Deletion from a doubly linked list

- Given a node to be deleted, t
  - Set t -> next -> prev to t -> prev
  - Set t -> prev -> next to t -> next
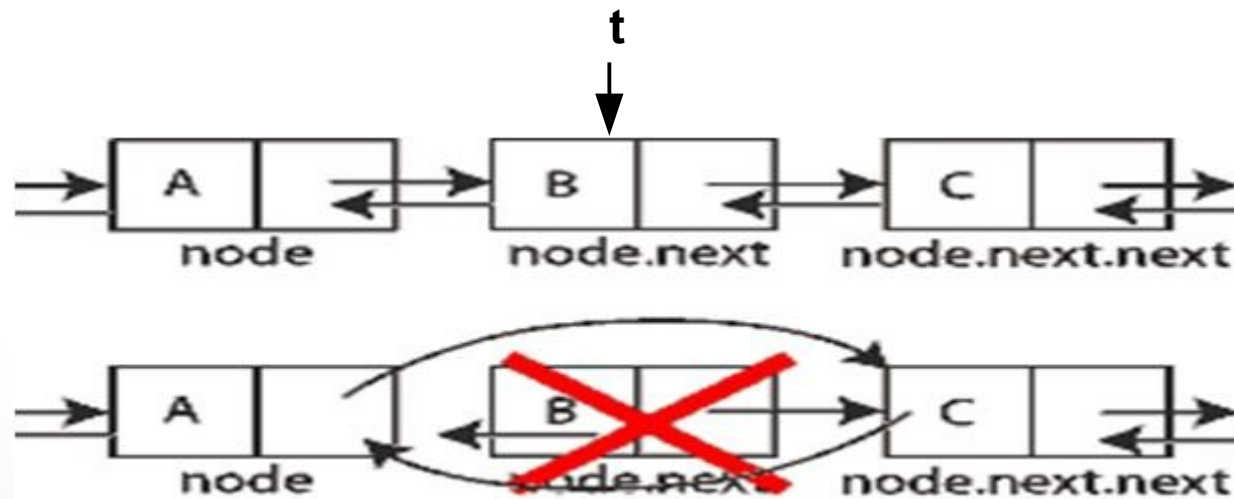
Insertion in a doubly linked list - Illustration
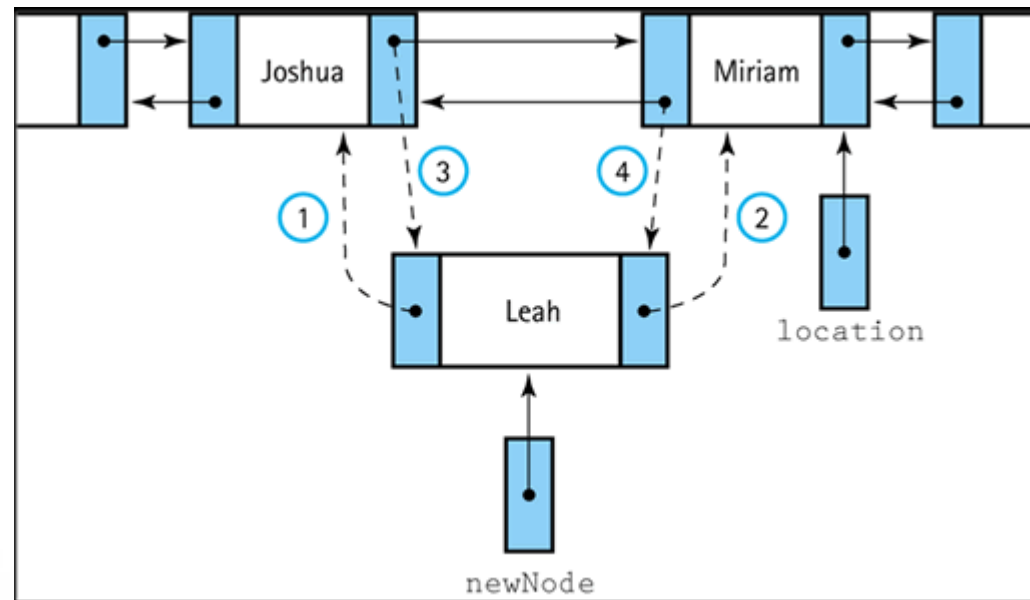
Insertion in a doubly linked list

- Inserting node t after node x

  - Set t->next to x->next

  - Set x->next->prev to t

  - Set x->next to t

  - Set t->prev to x
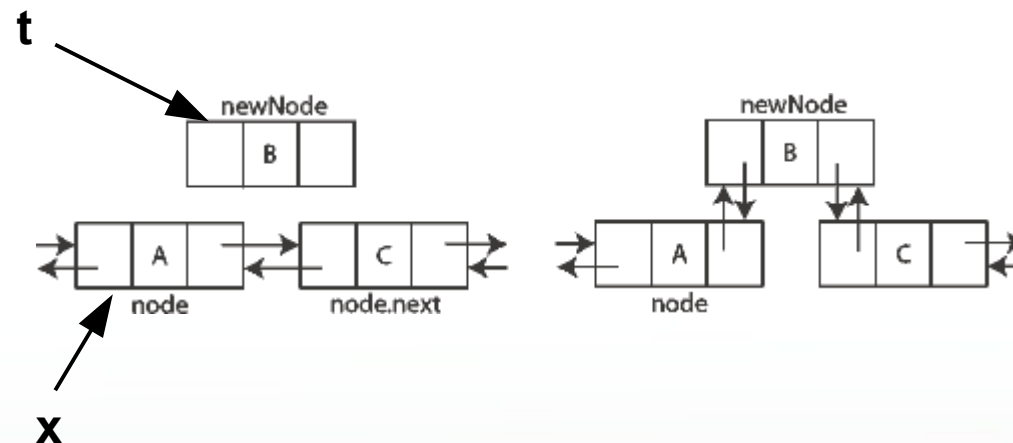
Circular List

- Last node is linked back to the first

  - Start with a single node linked to itself

 - For each additional node, readjust links accordingly
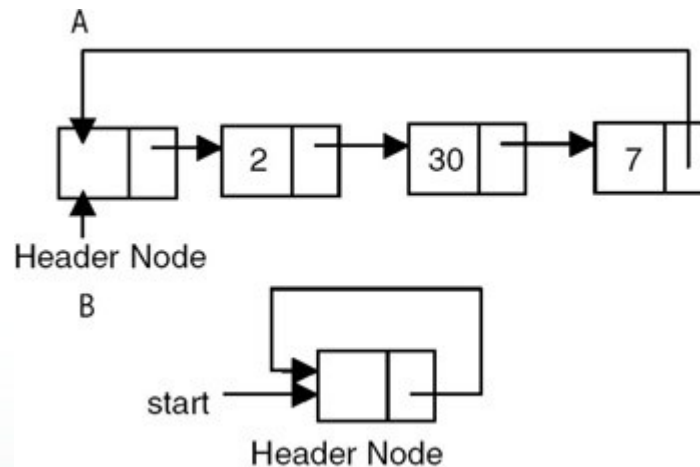
Josephus problem

- Phrased in various ways

  - Finding out which prisoner should escape execution

    - Prisoners arranged in a circle
    - Executioner walks along the circle, starting from a predetermined prisoner, removing every k-th prisoner and killing him
    - As the process goes on, the circle becomes smaller and smaller, until only one prisoner remains, who is then freed.

  - Electing a leader

    - Have a group of people stand in a circle/or sit round a conference table
    - Starting at a predetermined person, you count around the circle n times
    - Once you reach the nth person, take them out of the circle and have the members close the circle
    - Then count around the circle the same n times and repeat the process, until only one person is left. That person wins the election.

Josephus Problem Illustrated
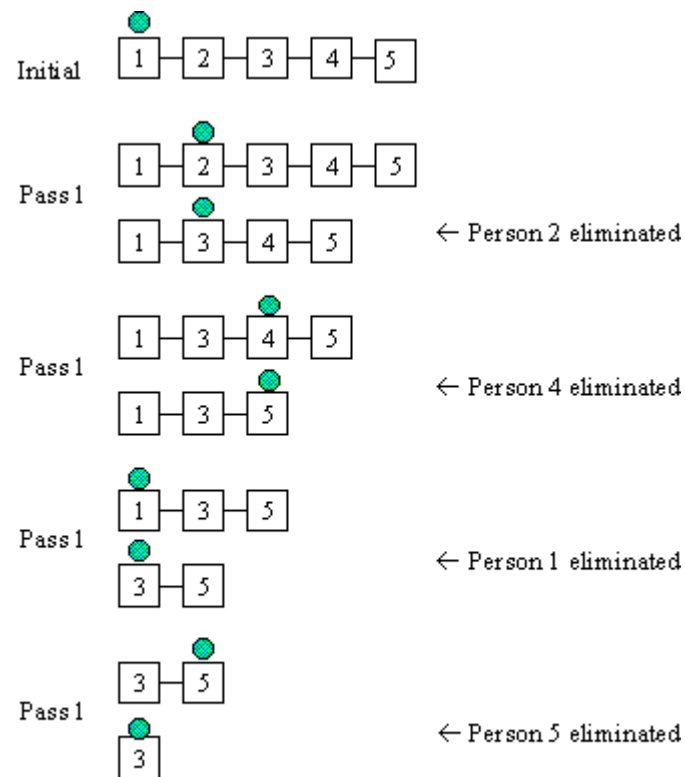
Josephus Problem Illustrated
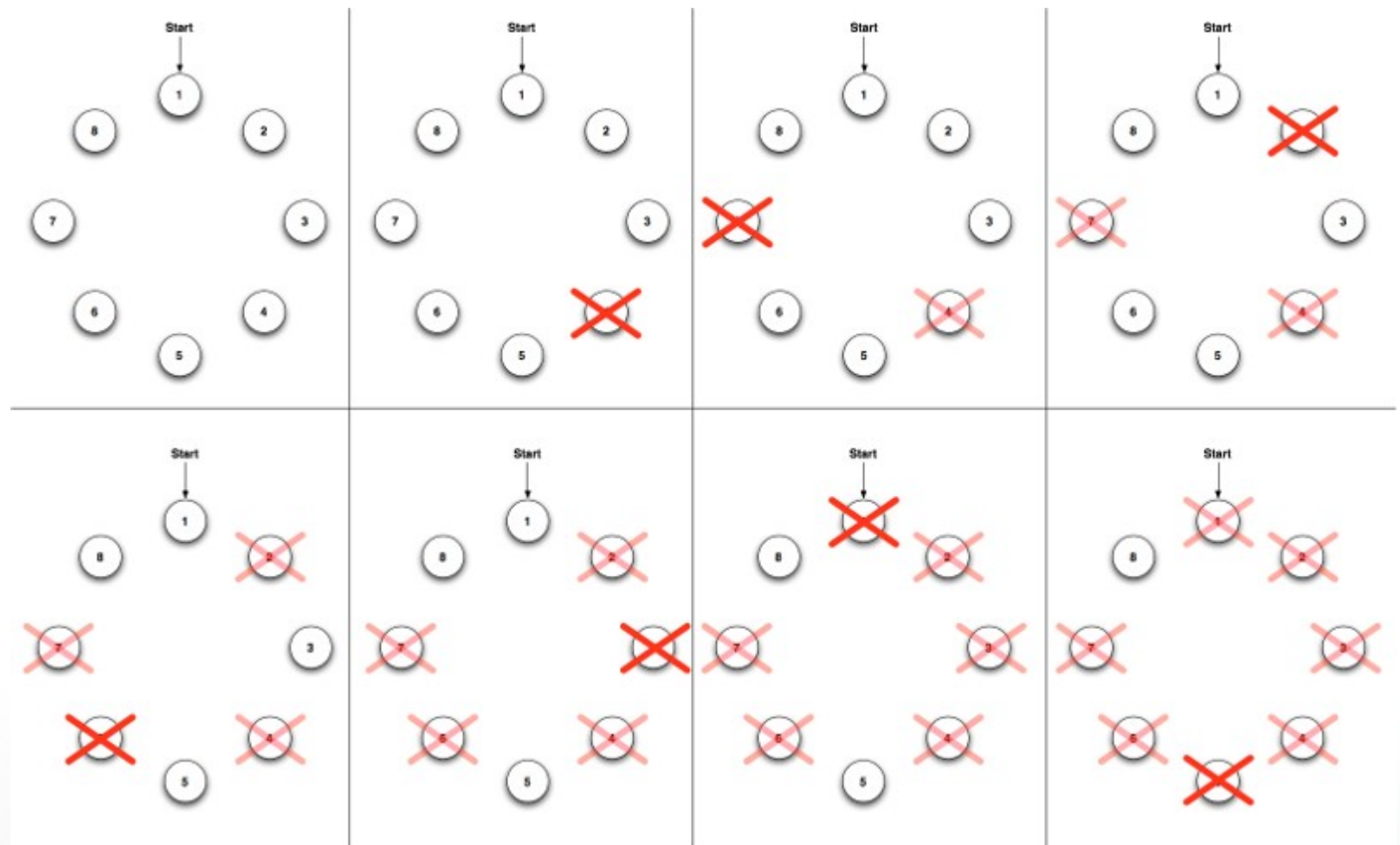
Josephus Problem

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct node * Link;
struct node { int item; Link next; };
int main(int argc, char** argv) {
    int i, N = atoi(argv[1]), M = atoi(argv[2]);
    Link t = malloc(sizeof *t), x = t;
    t -> item = 1; t-> next = t;     //1-node circular linked list
    for (i=2; i<=N; i++) {           //rest of nodes in circular list
        x = (x -> next = malloc(sizeof *x));
        x -> item = i;   x -> next = t;
    }
    x=t;
    while (x->next != t) {
        printf("%d ",x->item); x = x-> next;
    }
    printf("%d ",x->item);
    printf("\n");
    printf("Nodes eliminated: "); //implement josephus prolem below
    x = t;
    while (x != x-> next) {    //loop for as long as > 1 node available
        for (i=1; i<M; i++)    //skip M-1 nodes; we shall bypass Mth node
            x = x-> next;
        printf("%d ",x->next->item);
        x -> next = x->next->next;  N--;//printf("%d\n", x-> item);
    }
    printf("\nWinner is %d\n", x-> item);
}
```

Josephus Problem – Sample runs

```
josephus 9 5
1 2 3 4 5 6 7 8 9
Nodes eliminated: 6 2 8 5 4 7 1 3
Winner is 9

josephus 20 3
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Nodes eliminated: 4 7 10 13 16 19 2 6 11 15 20 5 12 18 8 17 9 3 14
Winner is 1
```
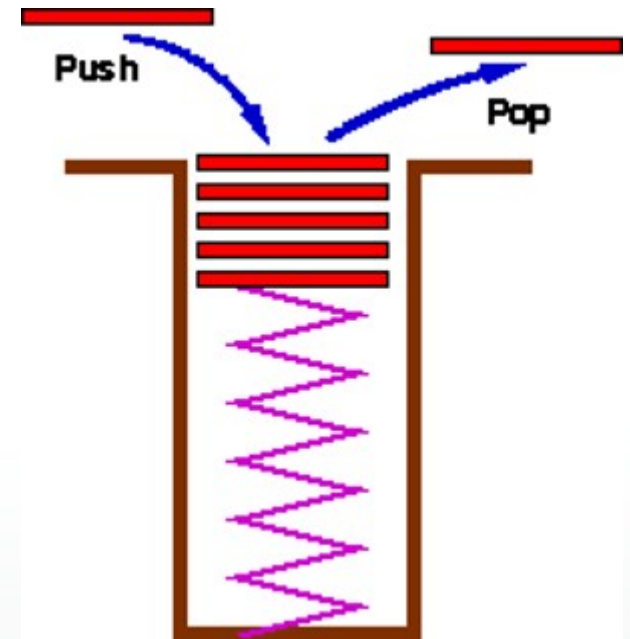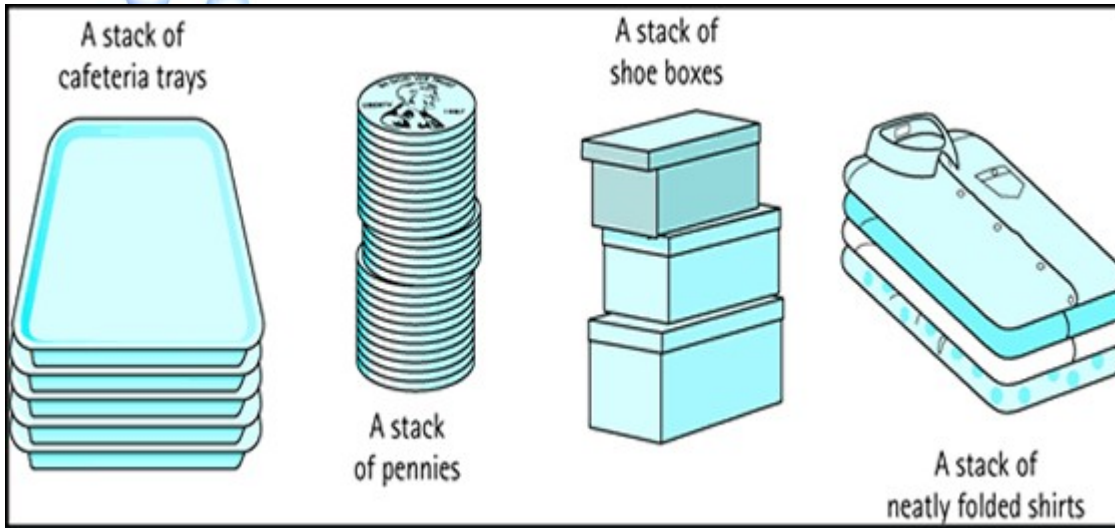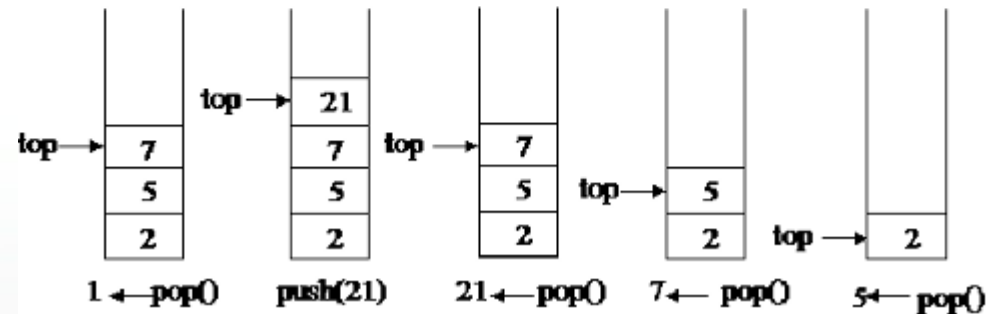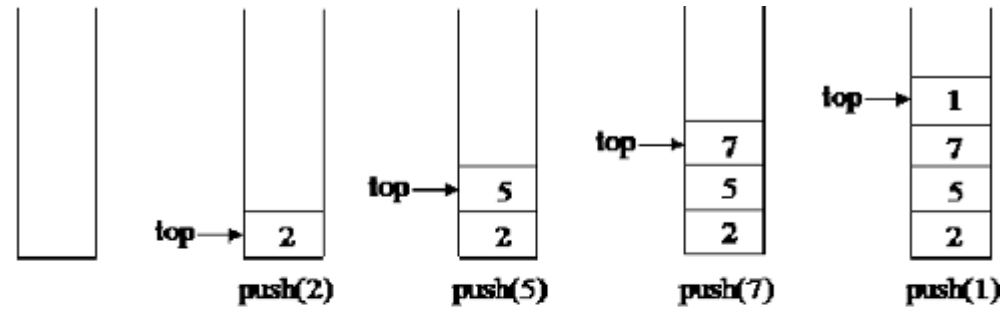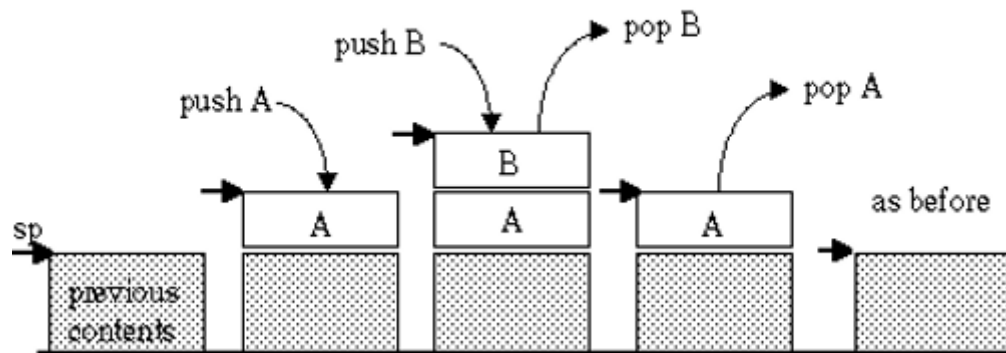
Stacks and Queues

- Stack
  - A last-in, first-out (LIFO) data structure
    - First item taken from stack is the last item that was placed there
  - Only accessible from the top of the stack
  - Analogy
    - Stack of plates in a restaurant
  - Computer programs are naturally organized in the form of stacks
    - Tasks are usually postponed (cf. the plates at the bottom of the stack) while performing other tasks (cf. using the plates that were recently put on the stack)
    - Moreover, the program usually needs to return to the most recently postponed tasks first
  - Stack operations
    - Push: Inserting an item on the stack
    - Pop: Taking an item off a stack
  - Tests on a stack
    - Stack-Empty: Used before pop() to ensure stack is empty
    - Stack-Full: Used before push() to ensure there is room in the stack for new item

Stack Illustrations

Stack Illustrations

Stack is LIFO

Stacks and queues

- Queue
   - First-in first-out (FIFO) data structure
        - Items are taken off the queue in the order in which they arrive the queue
        - Analogy
            - People waiting to get on a bus, or to get to supermarket checkout
        - Items taken off the head/front of the queue, and added to the tail/back of the queue
        - Queue operations
            - Enqueue: add item to the tail of a queue
            - Dequeue: remove item from the head of a queue
        - Queues useful in computers, example
            - Multi-user computer system for example
            - User tasks performed on a first-come first-served basis
            - Various tasks can be placed on a queue, and the operating system simply works on the task that has been in the queue longest
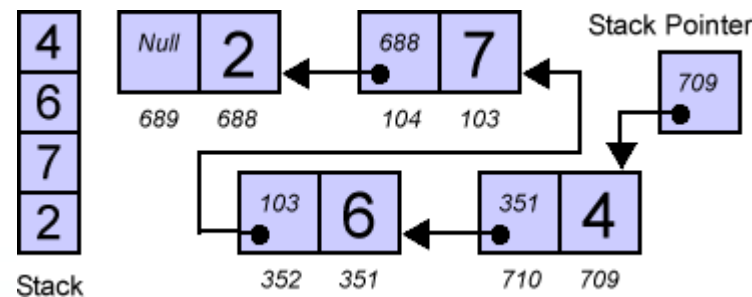
Queue Illustrations
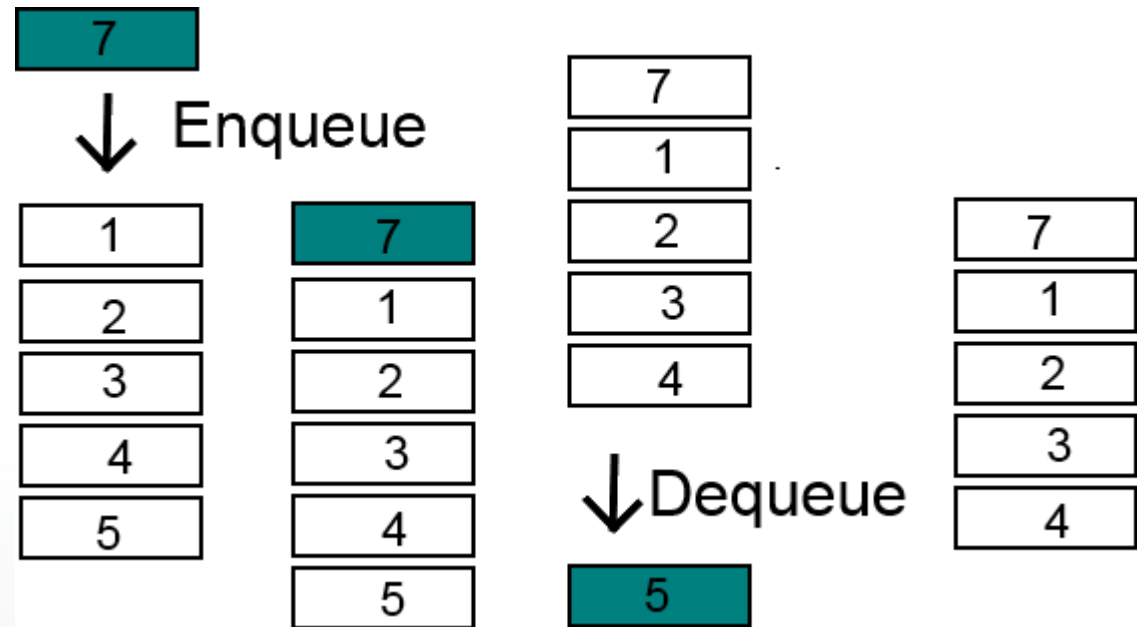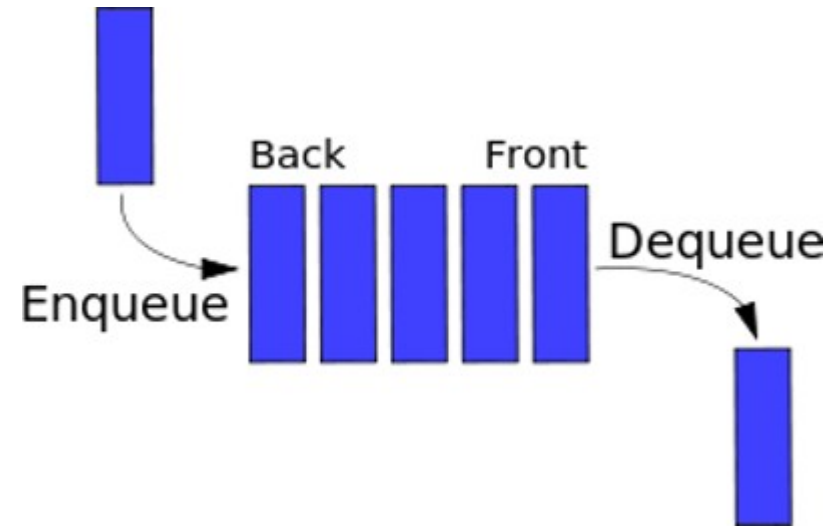
Implementing a stack (using an array)

- Use an array of size N

  - All operations happen at the stack pointer (top of stack)

  - Each time an item is added to the stack (PUSH), the stack pointer index is increased by one

  - When an item is returned from the stack (POP), the value of stack pointer index decreases by 1

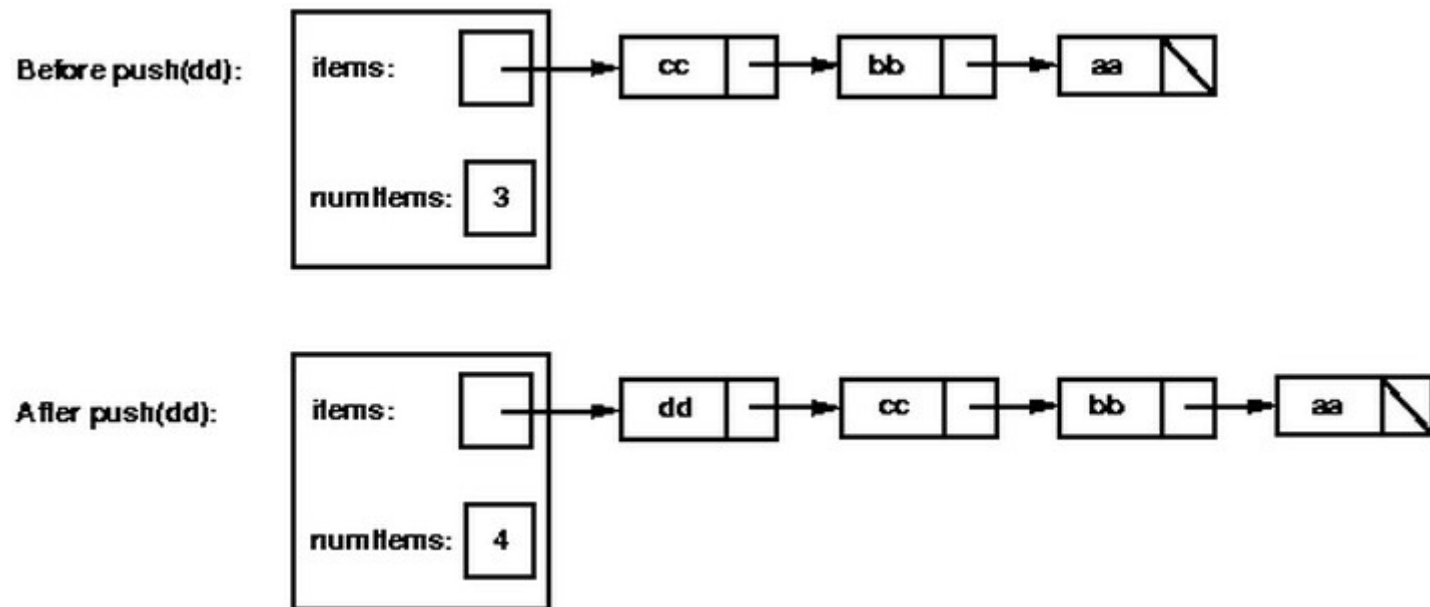   - No need to explicitly delete the popped item from the stack
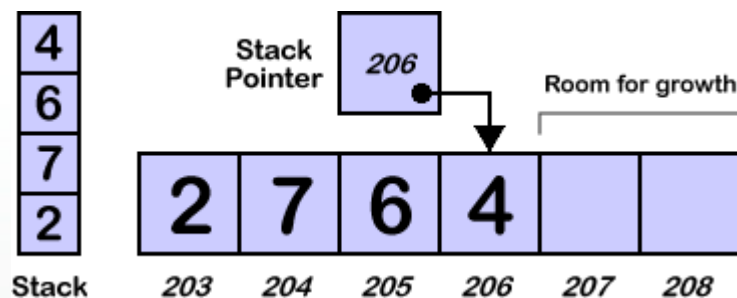
Implementing a stack (using a linked list)

- Ensuring that
  - All new items are added to the start of the list
 - All deletions are done at the start of the list

Implementing a stack (using a linked list)

Before push(dd):

items: → cc → bb → aa

numItems: 3

After push(dd):

items: → dd → cc → bb → aa

numItems: 4

Implementing a queue (using an array)
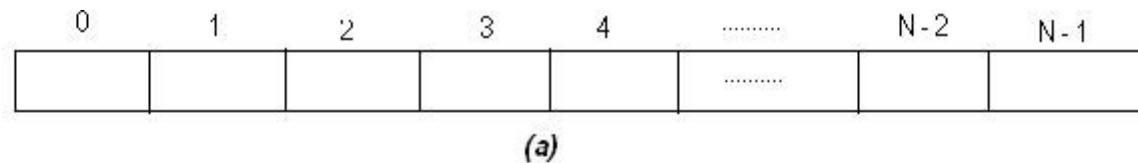
- Use an array of size N
  - Index 0 of the queue immediately follows location N-1 in a circular order.
  - When front index = rear index, the queue is empty
    - An attempt to dequeue an empty queue leads to an underflow error
  - Initially, front = rear = 0
  - When front = (rear + 1) mod N, the queue is full
    - An attempt to enqueue a queue that is full leads to an overflow error
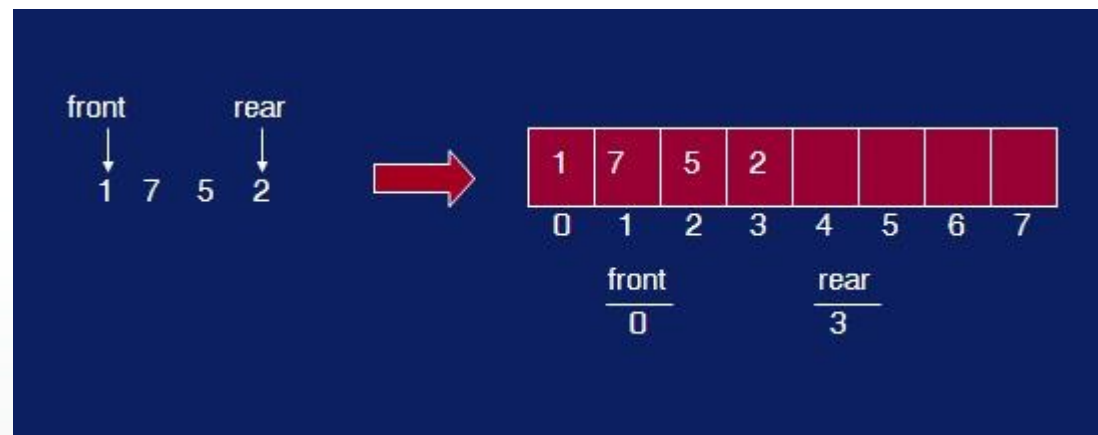
Implementing a queue (using an array)

front = NULL
rear = NULL

0	1	2	3	4	....	N-2	N-1

(a)

front = 0
rear = 0

0	1	2	3	4	....	N-2	N-1
6							

↑ front

(b)

front = 0
rear = 1

0	1	2	3	4	....	N-2	N-1
6	4						

↑ front   ↑ rear

(c)

front = 0
rear = 2

0	1	2	3	4	....	N-2	N-1
6	4	19					

↑ front   ↑ rear

(d)
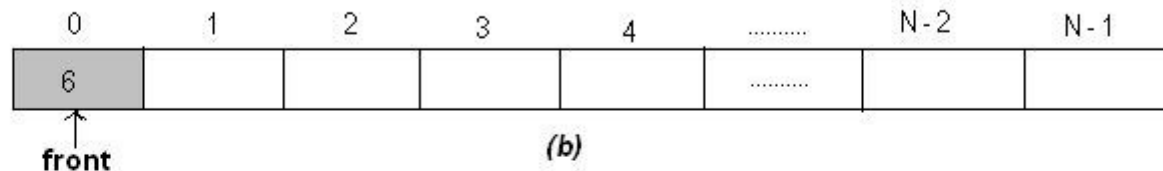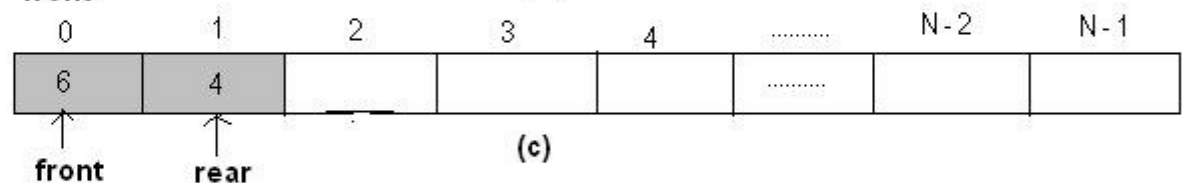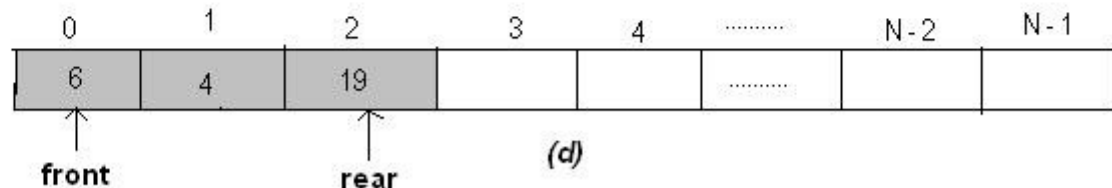
Insertion in a queue implemented as an array

Implementing a queue (using a linked list)

- Front/head pointer is maintained to ensure that dequeue only takes place at the head of the list

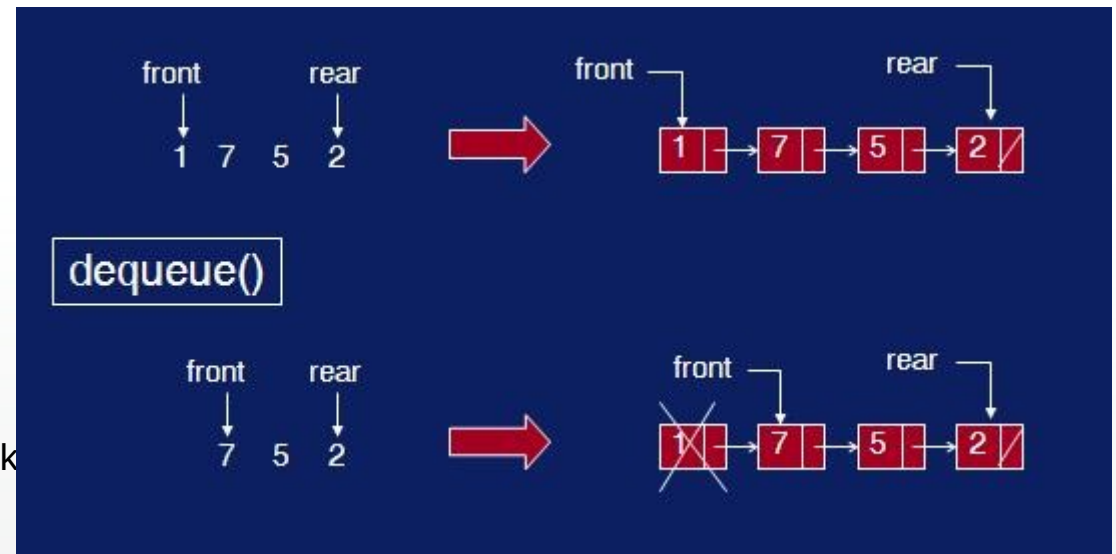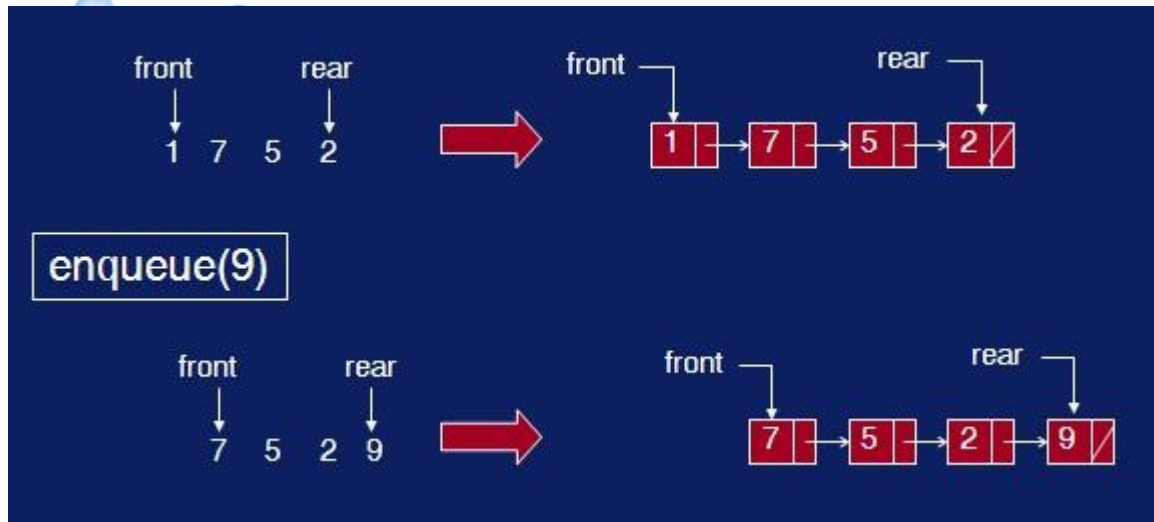- Tail pointer maintained to ensure that enqueue only takes place at the end of the list

D.L. Nk

Array implementation of stack

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 100 /*max size of stack*/
int *s; /*stack to contain integers*/
int N; /*top of stack*/
void stackInit(int max) {
  s = malloc(max*sizeof(int));  N = 0;
}
int stackEmpty() {
  return N == 0;
}
int stackFull() {
  return N == MAX;
}
void stackPush(int item) {
  s[N++] = item;
}
int stackPop(void) {
  return s[--N];
}
void stackDisplay(void) {
  int i;
  for (i = N-1; i >= 0; i--)
    printf("%d ",s[i]);
  printf("\n");
}
```

```c
int main(void) {
  int i;
  stackInit(MAX);
  /*add 10 random numbers to stack*/
  for (i = 0; i < 10; i++)
    if (!stackFull())
      stackPush(rand() % 1000);
  printf("Initial Contents: ");
  stackDisplay();
  /*pop 5 items off the stack*/
  for (i = 0; i< 5; i++)
    if (!stackEmpty())
      stackPop();
  printf("Contents after poping off 5
         items: ");
  stackDisplay();
  }
Initial Contents: 510 717 701 91 355 728
949 362 609 772
Contents after poping off 5 items: 728
949 362 609 772
```

Linked list implementation of stack

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 100 /*maximum stack size*/
typedef struct stackNode *LINK;
struct stackNode {
    int item; LINK next;
};
LINK head;
LINK newNode(int item, LINK h) {
  LINK nnode = malloc(sizeof(*nnode));
  nnode->item=item; nnode->next=h;
  return nnode;
}
void stackInit(int max) {head=NULL; }int
stackEmpty() {return head==NULL;}
void stackPush(int item) {
  head = newNode(item, head);
  printf("%d added to stack\n",item);
}
int stackPop(void) {
  int item=head->item;  LINK t=head->next;
  free(head); head = t;  return item;
}
void stackDisplay(void) {
  LINK t = head;
  while (t != NULL) {
    printf("%d ",t->item); t=t->next;
  }
  printf("\n");
}
```

```c
int main(void) {
  int i, item;
  stackInit(MAX);
  /*add 10 random numbers to stack*/
  for (i = 0; i < 10; i++)
    stackPush(rand() % 1000);
    printf("Initial Contents: ");
    stackDisplay();
    /*pop 5 items off the stack*/
    for (i = 0; i< 5; i++) {
      if (!stackEmpty()) {
        item = stackPop();
        printf("%d popped from stack\n",item);
      }
    }
    printf("Contents after popping items: ");
    stackDisplay();
}
```

Sample Output
772 added to stack 609 added to stack 362
added to stack 949 added to stack
728 added to stack 355 added to stack 91 added
to stack 701 added to stack
717 added to stack 510 added to stack
Initial Contents: 510 717 701 91 355 728 949
362 609 772
510 popped from stack 717 popped from stack
701 popped from stack
91 popped from stack 355 popped from stack
Contents after popping off 5 items: 728 949
362 609 772

Array implementation of queue

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 100 /*maxsize of queue*/
int *q; /*queue to contain integers*/
int N,head, tail;
void queueInit(int max) {
  q = malloc(max+1*sizeof(int));
  /*array size = queue size+1*/
  N = max+1; head = tail = 0;
}
int queueEmpty() {
  return head == tail;
}
int queueFull() {
  return head == (tail + 1) % N;
}
void queueDisplay(void) {
  int i;  int h = head;
  for (i = h; i != tail; i = h) {
     printf("%d ",q[i]); h=(h+1) % N;
  }
  printf("\n");
}
void enqueue(int item) {
  q[tail++] = item; tail = tail%N; }
```

```c
int dequeue(void) {
  int tmp = head; head = (head+1) % N;
   return q[tmp];
}
int main(void) {
  int i; queueInit(MAX);
  /*add 20 random numbers to queue*/
  for (i = 0; i < 20; i++)
    if (!queueFull())
      enqueue(rand() % 1000);
  printf("Initial Contents: ");
  queueDisplay();
  /*dequeue 10 items*/
  for (i = 0; i< 10; i++)
    if (!queueEmpty())
      dequeue();
  printf("Contents after dequeuing 5
        items: ");
  queueDisplay();
}
Initial Contents: 772 609 362 949 728
355 91 701 717 510 76 247 763 132 775
938 75 79 759 72
Contents after dequeuing 5 items: 76 247
763 132 775 938 75 79 759 72
```

Linked list implementation of queue

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 100 /*max size of queue*/
typedef struct queueNode *LINK;
struct queueNode {
int item; LINK next;
};
LINK head,tail;
LINK newNode(int item, LINK nxt) {
  LINK nnode = malloc(sizeof(*nnode));
  nnode -> item = item;
  nnode -> next = nxt;  return nnode;
}
void queueInit(int max) {head=NULL; }
int queueEmpty() {return head==NULL; }
void enqueue(int item) {
  if (head == NULL) {
    head = (tail=newNode(item,head));
    return;
  }
  tail->next=newNode(item,tail->next);
  tail = tail -> next;
}
int dequeue(void) {
  int item=head->item;LINK
t=head->next;
  free(head); head = t; return item;
}
```

```c
void queueDisplay(void) {
  LINK t = head;
  while (t != NULL) {
    printf("%d ",t -> item);t = t->next;
  }
  printf("\n");
}
int main(void) {
  int i, item;
  queueInit(MAX);
  /*add 10 random numbers to queue*/
  for (i = 0; i < 10; i++)
    enqueue(rand() % 1000);
  printf("Initial Contents: ");
  queueDisplay();
  /*dequeue 5 items off the queue*/
  for (i = 0; i< 5; i++) {
    if (!queueEmpty()) {
      item = dequeue();
    printf("%d dequeued from queue\n",
        item);
    }
  }
printf("Contents after dequeuing off 5
items: ");
queueDisplay();
}
```

Strings

- String

  - A variable length array of characters defined by a starting point and by a termination character

    - Not the same as array of characters

      - Character array has a fixed length

  - String terminating character in C is '\0' (the NULL character)

Some string processing functions

- strcat() - Link together (concatenate) two strings
- strncat() - Concatenate one string with part of another
- strchr() - Find character in string
- strrchr() - Find character in string (starting from right)
- strcmp() - Compare two strings.
- strncmp() - Compare parts of two strings
- strcpy() - Copy one string to another.
- strncpy() - Copy part of a string
- strlen() - Get length of a string.
- strlwr() - Convert string to lowercase.
- strupr() - Convert string to uppercase
- strstr() - Find string in string.

String processing example

```c
#include <stdio.h>
#define N 10000 /*assume max # of chars is 10000*/
int main(int argc, char *argv[]) {
    FILE *fp;
    char a[N];
    char *p = argv[1]; /*item to be searched for*/
    fp = fopen("ssearch.txt", "r");
    int c, i, j;
    for (i = 0; i < N-1; a[i] = c, i++) /*read data into array*/
        if ((c = getc(fp)) == EOF)
            break;
    a[i] = 0; /*string terminator*/
    /*now determine occurences of substring in array*/
    for (i = 0; a[i] != 0; i++) {
        for (j = 0; p[j] != 0; j++)
            if (a[i+j] != p[j])
                break;
        if (p[j] == 0)
        printf("%d ", i); /*print substring position*/
    }
    printf("\n");
    fclose(fp);
}
```

String processing example

File tower.txt

In the Indian city of Benares, beneath a dome that marked the center of the world, is to be found a brass plate in which are set three diamond needles, "each a cubit high and as thick as the body of a bee." Brahma placed 64 **disks** of pure gold on one of these needles at the time of Creation. Each **disk** is a different size, and each is placed so that it rests on top of another **disk** of greater size, with the largest resting on the brass plate at the bottom and the smallest at the top. Within the temple are priests whose job it is to transfer all the gold **disks** from their original needle to one of the others, without ever moving more than one **disk** at a time. No priest can ever place any **disk** on top of a smaller one, or anywhere else except on one of the needles. When the task is done, and all 64 **disks** have been successfully transferred to another needle, "tower, temple, and Brahmins alike will crumble into dust, and with a thunder-clap the world will vanish." The prediction (thunder-clap aside) seems fairly safe given that the number of steps required to transfer all the **disks** is $2^{64} - 1$, which is approximately 1.8447×10^{19}. Assuming one second per move, this would take about five times longer than the current age of the universe!

hanoi disk
227 300 380 560 649 694 805 1086